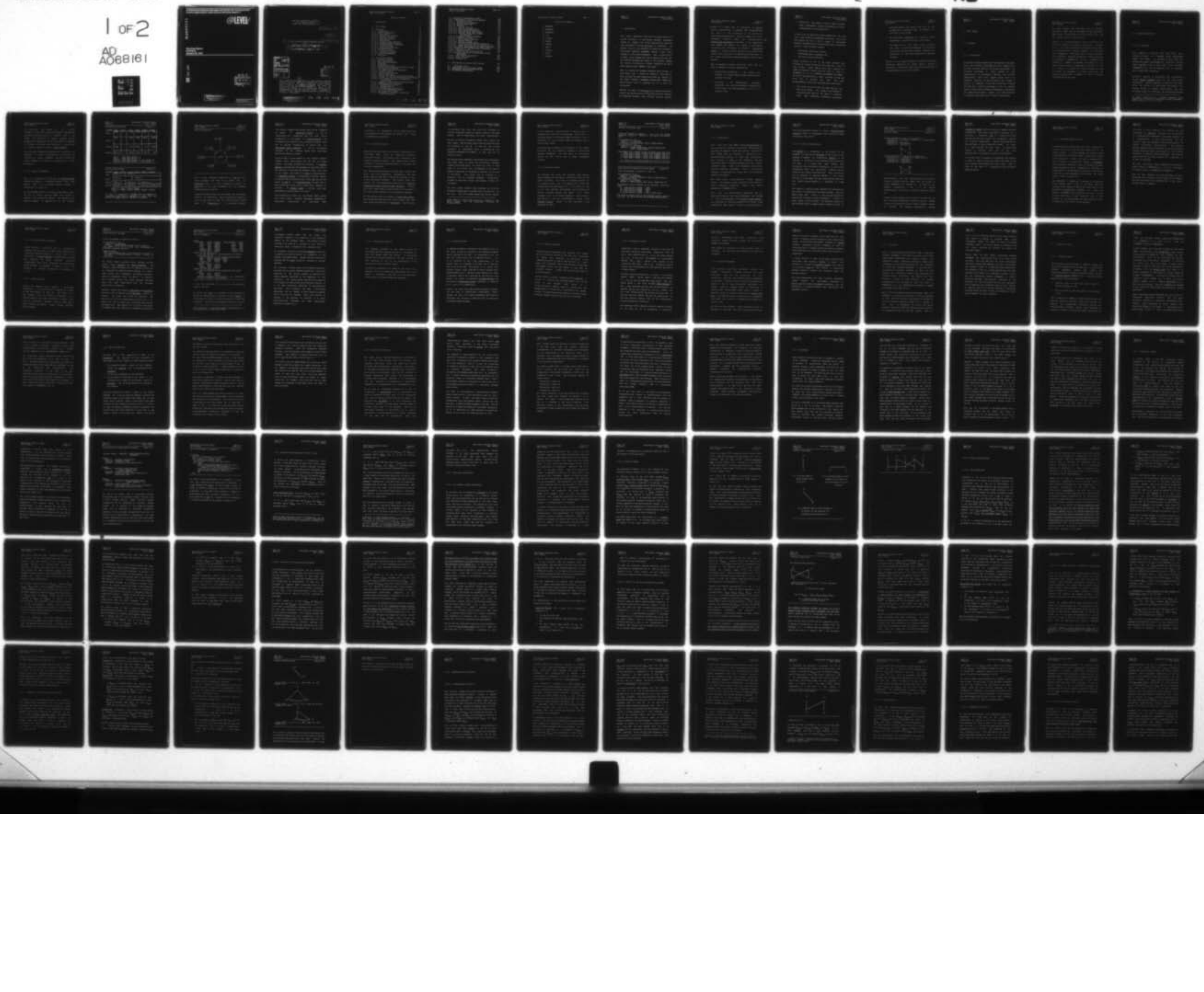
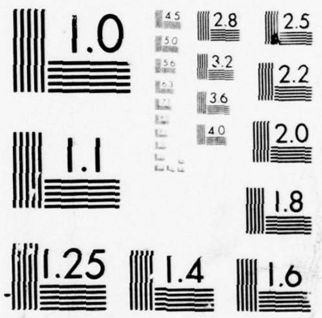
AD-A068 161 COMPUTER CORP OF AMERICA CAMBRIDGE MASS F/G 9/2
A DISTRIBUTED DATABASE MANAGEMENT SYSTEM FOR COMMAND AND CONTRO--ETC(U)
JAN 79 N00039-77-C-0074
UNCLASSIFIED CCA-79-12 NL

1 OF 2
AD
A068161




1.0
4.5
5.0
5.6
6.3
7.1
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

A Distributed Database Management System for Command and Control Applications: Semi-Annual Technical Report 4

Technical Report
CCA-79-12
January 30, 1979

Computer Corporation of America
575 Technology Square
Cambridge, Massachusetts 02139

Computer Corporation of America
575 Technology Square
Cambridge, Massachusetts 02139

#3 - A060441

(11) 3∅ Jan 79

(12) 152p.

(14) CCA-79-12

(6) A Distributed Database Management System for Command and Control Applications.

(9) SEMI-ANNUAL TECHNICAL REPORT IV no. 4,

1 July ~~1, 1978 to~~ December (31), 1978,

(15)

This research was supported by the Defense Advanced Research Project Agency of the Department of Defense and was monitored by the Naval Electronic System Command under Contract No. N00039-77-C-0074. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

387 285

79 03 16 053

sat

Table of Contents

Project Staff Members:

T. ANDERSON

B. BERKOWITZ

P. BERNSTEIN

S. FOX

N. GOODMAN

M. HAMMER

T. LANDERS

C. REEVE

J. ROTHNIE

S. SARIN

D. SHIPMAN

1. Introduction

This report summarizes the fourth six month period of a project entitled, "A Distributed Database Management System for Command and Control Applications" which has been undertaken by CCA and sponsored by ARPA-IPTO. The The primary focus of this ~~effort~~ project is to design and implement a distributed database management system called SDD-1 (System for Distributed Databases). SDD-1 is specifically oriented toward command and control applications and will be installed ~~in phases~~ and tested in the Advanced Command and Control Architectural Testbed ~~(ACCAT)~~ at the Naval Ocean Systems Center ~~(NOSC)~~ in San Diego.

SDD-1 is a system for managing databases whose storage is distributed over a network of computers. Functionally, SDD-1 provides the same capabilities that one expects of any modern database management system, ~~(abbr. DBMS)~~ and users interact with it precisely as if it were not distributed.

Systems like SDD-1 are appropriate for applications which exhibit two characteristics: (1) ~~First,~~ the activity requires an integrated database~~. That is,~~ i.e. the activity entails access to a single pool of information by multiple persons, organizations, or programs; And ~~second,~~ (2) either the users of the information or its sources are distributed geographically. Military command and control ~~obviously~~ exhibits these two characteristics. Decentralized processing is desirable in an application like command and control for reasons of performance, reliability, and flexibility of function. Centralized control is needed to ensure operation in accordance with overall policy and goals. By meeting both these goals in one system, distributed database management offers unique benefits.

SDD-1 is designed to achieve three other goals that are vital to command and control applications:

1. Reliability/survivability - the system will continue to operate despite communications and/or processor failures.

2. Efficiency - the communications bottleneck associated with centralized DBMSs is minimized by storing data in the same geographical area it is primarily used.

3. Scalability - the system can grow to meet increased usage requirements without the necessity of major reconfiguration of existing sites.

Up until now, no system with these capabilities has been built even though the advantages are obvious. One of the main reasons for this is that a number of challenging technical problems must be solved before such a system can be built. These problems include:

- distributed concurrency control;
- distributed query processing; and
- achieving reliable operation.

Design solutions to all of the above problems were produced and refined during the first year and a half of this project. These results have been reported in a number of technical reports ([CCA a], [CCA b], [CCA c], [BERNSTEIN et al. b], [ROTHNIE and GOODMAN] and [WONG]). An initial implementation of SDD-1 occurred during the first half of calendar year 1978. The major accomplishments during this reporting period have been:

1. The initial version of SDD-1 has been improved and made more robust. The system is now capable of running multiple users at the same or different sites. Some rudimentary reliability mechanisms

have been added to the system such that it can automatically reconfigure when it detects or is told that a site is down.

2. The concurrency mechanisms are currently being implemented in the next version of the system.

3. All aspects of the major SDD-1 design solutions have been or are being documented in a new series of technical reports that are being submitted for publication in TODS (ACM Transactions on Database Systems).

Section 2 of this report presents an overview of the SDD-1 design and summarizes the design results. Section 3 discusses in detail the recent work that has been done on the implemented version of the system.

2. SDD-1 Design

2.1 Overview

2.1.1 Introduction

Distributed database systems pose new technical challenges due to their inherent requirements for data communication and their inherent potential for parallel processing. The principal bottleneck in these systems is data communication. All economically feasible long distance communication media incur lengthy delays and/or low bandwidth. Moreover, the cost of moving data through a network is comparable to the cost of storing it locally for many days. Parallel processing is also an inherent aspect of distributed systems and mitigates to some extent the communication factor. However, it is often difficult to construct algorithms that can exploit parallelism.

For these reasons, the techniques used to implement centralized DBMSs must be re-examined in the distributed DBMS context. We have done this in developing SDD-1 and this section surveys our main results.

Section 2.1.2 describes SDD-1's overall architecture and the flow of events in processing transactions. Sections 2.1.3 - 2.1.5 then introduce the techniques used by SDD-1 for solving the most difficult problems in distributed data management: concurrency control, query processing, and reliability. Detailed discussions of these techniques are presented in ⌊BERNSTEIN et al. a,b⌋, ⌊BERNSTEIN and SHIPMAN a⌋, ⌊HAMMER and SHIPMAN⌋, and ⌊WONG et al.⌋ and some of these results are summarized in sections 2.2 and 2.3. Section 2.1.6 explains how these techniques are used to handle the management of system directories. The section concludes with a summary of SDD-1's principal contributions to the field.

2.1.2 System Organization

2.1.2.1 Data Model

SDD-1 supports a relational data model [CODD]. Users interact with SDD-1 in a language called Datalanguage [MARILL and STERN]. For purposes of this report, the differences between Datalanguage and relational calculus based languages such as QUEL [HELD et al.] or SEQUEL [CHAMBERLIN et al.] are not important, and for pedagogic ease we adopt QUEL terminology.

Each SDD-1 relation is partitioned into sub-relations called logical fragments which are the units of data distribution. Logical fragments are defined in two steps. First, the relation is partitioned horizontally into subsets defined by "simple" restrictions.* Then each horizontal subset is partitioned into subrelations defined

---

*A simple restriction is a boolean expression whose clauses are of the form <attribute> <rel_op> <constant>, where <rel_op> is =, ≠, >, <, etc.

by projections. (See Figures 2.1, 2.2.) Logical fragments are the unit of data distribution, meaning that each may be stored at any one or several sites in the system. The definition of logical fragments and the assignment of fragments to sites occurs when the database is designed and remains fixed thereafter. A stored copy of a logical fragment is called a stored fragment.

Note that user transactions reference only relations, not fragments. It is SDD-1's responsibility to translate from relations to logical fragments, and then to select the stored fragments to access in processing any given transaction.

2.1.2.2 General Architecture

SDD-1 is a collection of three types of virtual machines [HORNING and RANDELL] -- Transaction Modules (TMs), Data Modules (DMs), and a Reliable Network (RelNet) -- configured as in Figure 2.3.

All data managed by SDD-1 is stored by Data Modules (DMs). DMs are, in effect, back-end DBMSs that respond to commands from Transaction Modules. DMs respond to four types of commands: (1) Read part of the DM's database

Horizontal Partitioning

Figure 2.1

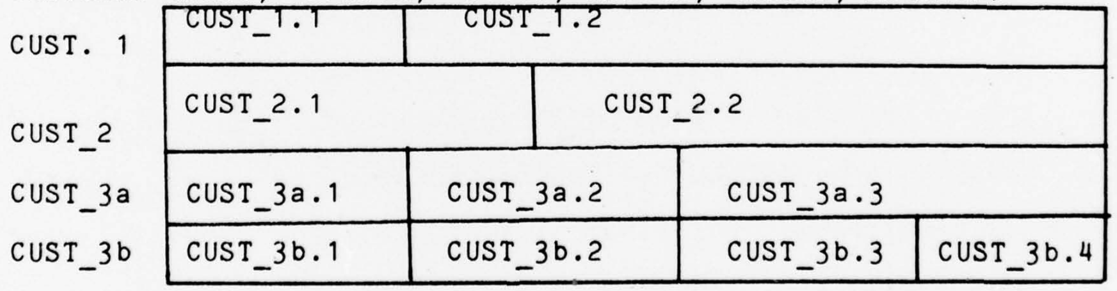
CUSTOMER	(Name,	Branch,	Acct#,	SavBal,	ChkBal,	LoanBal)
CUST_1	Wash. . .	1	1234	$100	$200	-$8
CUST_2	Jeff.	2	5678	$200	$300	$30000
CUST_3a	Adams	3	9012	$1000	$0	$20000
CUST_3b	Munroe	3	3456	$100	$50	$0

```
CUST_1  = CUST where Branch = 1
CUST_2  = CUST where Branch = 2
CUST_3a = CUST where Branch = 3 and LoanBal ≠0
CUST_3b = CUST where Branch = 3 and LoanBal = 0
```

Vertical Partitioning

Figure 2.2

CUSTOMER (Name, Branch, Acct#, SavBal, ChkBal, LoanBal)

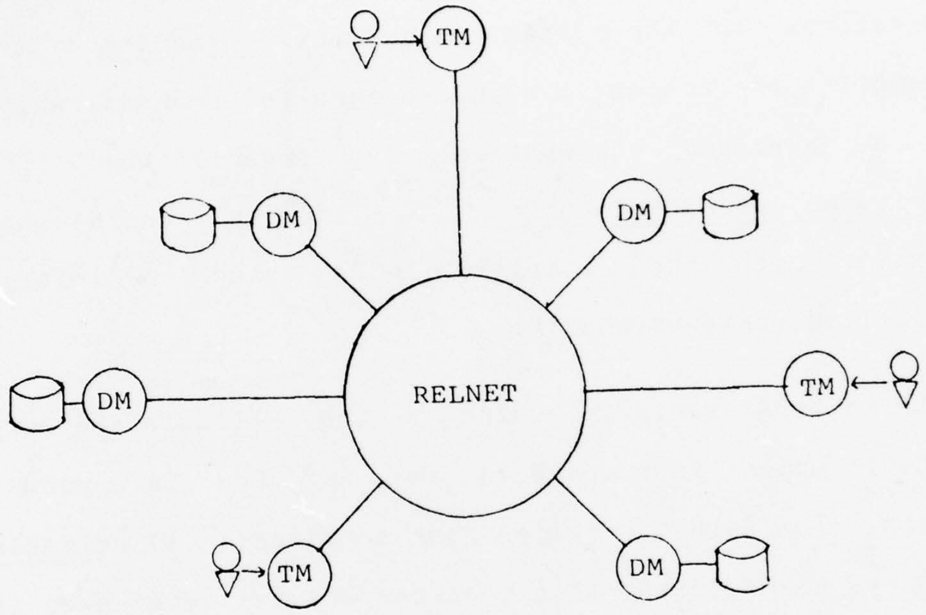


```
CUST_1.1 = CUST_1 [Name, Branch]
CUST_1.2 = CUST_1 [Acct#, SavBal, ChkBal, LoanBal]
  etc.
```

In order to reconstruct CUSTOMER from its fragments, a unique tuple identifier is appended to each tuple and included in <u>every</u> fragment [ROTHNIE and GOODMAN].

SDD-1 Configuration

Figure 2.3



into a local workspace at that DM; (2) Move part of a local workspace from this DM to another DM; (3) Manipulate data in a local workspace at the DM; (4) Write part of the local workspace into the permanent database stored at the DM.

Transaction Modules (TMs) plan and control the distributed execution of transactions. Each transaction processed by SDD-1 is supervised by some TM, which performs these tasks: (1) Fragmentation -- it translates relations into logical fragments and decides which stored fragments to access. (2) Concurrency control -- the TM synchronizes the transaction with all other active transactions in the system. (3) Access planning -- the TM compiles the transaction into a parallel program which can be executed cooperatively by several DMs. (4) Distributed query execution -- the TM coordinates execution of the compiled access plan, exploiting parallelism whenever possible.

The third SDD-1 virtual machine is the Reliable Network (RelNet) which interconnects TMs and DMs in a robust fashion. The RelNet provides four services: (1) Reliable delivery, guaranteeing that messages are delivered even if the recipient is down when the message is sent , and even if the sender and receiver are never up simultaneously. (2) Transaction control, a mechanism for posting updates at multiple DMs, guaranteeing that either all DMs post the update or none do. (3) Site monitoring, to keep track of which sites have failed, and to inform sites impacted by failures. (4) Network clock, a virtual clock kept approximately synchronized at all sites.

This architecture divides the distributed DBMS problem into three pieces: database management, management of distributed transactions, and distributed DBMS reliability. By implementing each of these pieces as a self-contained virtual machine, the overall SDD-1 design is substantially simplified.

2.1.2.3 Run-Time Structure

Among the functions required to execute a transaction in a distributed DBMS, three are especially difficult: concurrency control, distributed query processing, and reliable posting of updates. SDD-1 handles each of these problems in a distinct processing phase, so that each can be solved independently.

The three processing phases are called Read, Execute, and Write. Let T be a transaction. Conceptually, the Read phase reads all data that T references and places it in a private distributed workspace. The Execute phase then performs the data manipulation specified by T, _doing all reading and writing in that private workspace_. Finally, the Write phase takes all data written by T and moves it from the private workspace into the permanent database.

The Read phase exists for purposes of _concurrency control_. Using mechanisms described later, SDD-1 ensures that data read during the Read phase is _consistent_. Since the data is consistent when read, and since the workspace is private, subsequent phases can operate freely on this data without fear of interference from other transactions.*

No data is actually transferred between sites during the Read phase. Each DM simply sets aside the specified data in a workspace at the DM. In each DM, the private workspace is implemented using a differential file mechanism [SEVERANCE and LOHMAN], so data need not be actually copied.

The Execute phase implements distributed query processing. This phase takes as input the distributed workspace created by the Read phase. Its output is a list of data items to be written into the database (in the case of update transactions) or displayed to the user (in the case of retrievals). This output list is produced in a workspace, not the permanent database. Consequently, problems of concurrency control and reliable writing are irrelevant.

The Write phase installs data modified by T into the permanent database and/or displays data retrieved by T to the user. The Write phase ensures that partial results

---

*Some aspects of concurrency control are handled by the Write phase, but the mechanisms involved are straightforward.

are not installed or displayed even if multiple sites or communication links fail in mid-stream. This is the most difficult aspect of distributed DBMS reliability, and by separating it into a distinct phase, we simplify both it and the other phases.

The three-phase processing of transactions in SDD-1 neatly partitions the key technical problems of distributed database management. The next parts of this section explain how SDD-1 solves each of these independent problems.

2.1.3 Concurrency Control

The problems that arise when multiple users access a shared database are well-known. Generically there are two types of problems: (1) If user R is reading a portion of the database while user U is updating it, R might read inconsistent data (see Figure 2.4). (2) If users U1 and U2 are both updating the database, race conditions can produce erroneous results (see Figure 2.5). These problems arise in all shared databases -- centralized or distributed -- and are conventionally solved using database locking. However, we have developed a new technique for SDD-1.

Reading Inconsistent Data   Figure 2.4

Given the database of Figures 2.1 and 2.2, and assume fragments CUST_3a.2, CUST_3a.3, are stored at different DMs:

```
Let transaction R be
   Range of C is CUSTOMER;
   Retrieve C (SavBal+ChkBal) where C.Name="Adams";
Let transaction U be
   Range of C is CUSTOMER;
   Replace C (SavBal=SavBal-$100, ChkBal=ChkBal+$100)
              where C.Name="Adams";
```

And suppose R & U execute in the following concurrent order

```
   R  reads Adam's SavBal (=$1000) from fragment CUST_3a.2
   U writes Adam's SavBal (= $900) into fragment CUST_3a.2
   U writes Adam's ChkBal (= $100) into fragment CUST_3a.3
   R  reads Adam's ChkBal (= $100) from fragment CUST_3a.3
```

R's output will be $1000+$100=$1100, which is incorrect.

Race Condition Producing Erroneous Update   Figure 2.5

Given the database of Figures 2.1 and 2.2.

```
Let transaction U1 be
   Range of C is CUSTOMER;
   Replace C (ChkBal=ChkBal+$100) where C.Name="Munroe";
Let transaction U2 be
   Range of C is CUSTOMER;
   Replace C (ChkBal=ChkBal- $50) where C.Name="Munroe";
```

And suppose U1 and U2 execute in the following concurrent order

```
   U1  reads Munroe's ChkBal   (=$50)
   U2  reads Munroe's ChkBal   (=$50)
   U2 writes Munroe's ChkBal   (=$0 )
   U1 writes Munroe's ChkBal   (=$50 + $100= $150)
```

The value of Chk_Bal left in the database is $150, which is incorrect. The final balance should be $50-$50+$100=$100.

2.1.3.1 Methodology

SDD-1, like most other DBMSs, adopts serializability as its criterion for concurrent correctness. Serializability requires that whenever transactions execute concurrently, their effect must be identical to some serial (i.e., non-interleaved) execution of those same transactions. This criterion is based on the assumption that each transaction maps a consistent database state into another consistent state. Given this assumption, every serial execution preserves consistency. Since a serializable execution is equivalent to a serial one, it too preserves database consistency.

Most DBMSs ensure serializability through database locking. By locking, we mean a synchronization method in which transactions explicitly reserve data before accessing it [ESWARAN et al].

SDD-1 uses two synchronization mechanisms that are distinctly different from locking [BERNSTEIN and SHIPMAN b]. The first mechanism, called conflict graph analysis, is a technique for analyzing transactions to detect those transactions that require little or no synchronization.

The second mechanism consists of a set of synchronization protocols based on "timestamps", which synchronize those transactions that need it.

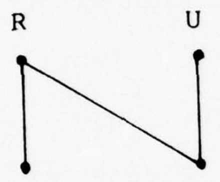
2.1.3.2 Conflict Graph Analysis

The read-set of a transaction is the portion of the database it reads and its write-set is the portion of the database it updates. Two transactions conflict if the read-set or write-set of one intersects the write-set of the other. In a system that uses locking, each transaction locks data before accessing it, so conflicting transactions never run concurrently. However, not all conflicts can violate serializability. More concurrency can be attained by checking whether or not a given conflict is troublesome, and only synchronizing those that are. Conflict graph analysis is a technique for doing this.

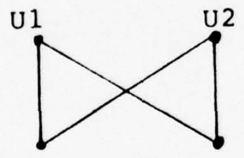
The nodes of a conflict graph represent the read-sets and write-sets of transactions, and edges represent conflicts among these sets. (There is also an edge between the read-set and write-set of each transaction.) Figure 2.6 shows sample conflict graphs. The important property is that different kinds of edges require different levels of

Conflict Graphs Figure 2.6

```
Define transactions R and U as in Figure 2.4
   read-set (R) = {C.SavBal, C.ChkBal s.t. C.Name="Adams"}
   write-set(R) = {}
   read-set (U) = read-set(R)
   write-set(U) = read-set(R)
```



```
Define transaction U1 and U2 as in Figure 2.5
   read-set (U1) = {C.ChkBal s.t. C.Name="Munroe"}
   write-set(U1) = read-set(U1)
   read-set (U2) = read-set(U1)
   write-set(U2) = read-set(U1)
```



synchronization, and that synchronization as strong as locking is required only for edges that participate in cycles [BERNSTEIN and SHIPMAN a]. In Figure 2.6, for example, transactions R and U do not require synchronization as strong as locking, whereas U1 and U2 do.

Conflict graph analysis could be used at run-time but too much inter-site communication would be required. Instead we apply the technique off-line, during database design, as follows: the database administrator defines transaction classes, which are groups of commonly executed transactions. Each class is defined by a read-set and a write-set; a transaction fits in a class if the class read-set and write-set contain the transaction read-set and write-set respectively. Conflict graph analysis is then performed on these transaction classes. The output is a table telling for each class: (a) which other classes it conflicts with, and (b) how much synchronization is required to ensure serializability.

At run-time, when a transaction is submitted, the TM finds a class in which it fits, and looks in the table to see how to synchronize transactions in that class. What it finds in the table is a composite of the "protocols" described below.

2.1.3.3 Timestamp Based Protocols

To synchronize two transactions that conflict dangerously, one must be run first, and the other delayed until it can safely proceed. In locking systems, the execution order is determined by the order in which transactions request conflicting locks. In SDD-1, the order is determined by a total ordering of transactions induced by _timestamps_. Each transaction submitted to SDD-1 is assigned a globally unique timestamp by its TM. Timestamps are generated by concatenating a TM identifier to the right of the network clock time, so that timestamps from different TMs always differ in their low order bits.

The timestamp of a transaction is attached to all Read and Write commands sent to DMs on its behalf. When a DM receives a Read command it defers the command until it has processed all _earlier_ Write messages (i.e., those with smaller timestamps) from a specified set of TMs. The set of TMs is specified by a "Read condition" that is also attached to the Read message; the Read condition in turn is specified by the conflict graph analysis.

Four kinds of Read conditions are possible and each corresponds to a <u>protocol</u>. A protocol is a specification of a synchronization requirement, and a Read condition is an implementation of that specification. The four protocols range from inexpensive local synchronization to complex distributed synchronization. The protocols prevent dangerous conflicts detected by the conflict graph analysis from occurring at run-time and do not directly correspond to familiar locking techniques. Each protocol ensures that all data read on behalf of a transaction at all DMs is consistent.

The SDD-1 concurrency control mechanism is described in greater detail in [BERNSTEIN et al. a,b]; its correctness is formally proved in [BERNSTEIN and SHIPMAN a].

When all Read commands have been processed, the TM is guaranteed that consistent, private copies of the read-set have been set aside at all necessary DMs. At this point, the Read phase is complete.

2.1.4 Distributed Query Processing

Having obtained a consistent copy of a transaction's read-set, the next step is to compile the transaction into a parallel program and execute it. The key part of the compilation is Access Planning, an optimization procedure that minimizes the object program's inter-site communication needs while maximizing its parallelism. Access Planning is discussed in Section 2.1.4.1, and execution of compiled transactions is explained in 2.1.4.2.

2.1.4.1 Access Planning

Perhaps the simplest way to execute a distributed transaction T is to move all of T's read-set to a single DM, and then execute T at that DM. (See Figure 2.7.) This approach works but suffers two drawbacks: (1) T's read-set might be very large, and moving it between sites could be exorbitantly expensive; and (2) little use is made of parallel processing. Access Planning overcomes these drawbacks.

Simple Execution Strategy Figure 2.7

Given the database of Figures 2.1 and 2.2

Let transaction T be
```
   Range of C is CUSTOMER;
   Replace C (ChkBal=ChkBal-LoanBal) where LoanBal<0;
```
(The effect of T is to credit loan overpayments to customers' checking accounts.)

Simple strategy
  Move every fragment that could potentially contribute to T's result to a designated site. Process T locally at that site.

The Access Planner produces object programs with two phases, called <u>reduction</u> and <u>final processing</u>. The reduction phase eliminates from T's read-set as much data as is economically feasible without changing T's answer. Then, during final processing, the reduced read-set is moved to a designated "final" DM where T is executed. This structure mirrors the simple approach described above, but lowers communication cost and increases parallelism via reduction.

Reduction employs the familiar <u>restriction</u> and <u>projection</u> operators, plus an operator called <u>semi-join</u>, defined as follows: let R(A,B) and S(C,D) be relations; the semi-join of R by S on a qualification q (e.g. R.B = S.C) equals the join of R and S on q, projected back onto the attributes of R. (See Figure 2.8.) If R and S are stored at different DMs, this semi-join is computed by projecting

---

Semi-Join Examples Figure 2.8

```
Given:
  CUST(Name,    ChkBal, LoanBal)   AUTO_PAY(Name,  Amount)
       Jeff.     $300    $30000             Jeff.    $300
       Adams     $100    $20000             Adams    $200
       Polk      $250    $20000             Polk     $200
       Tyler     $100    $15000             Tyler    $150
       Buchanan  $700    $40000             Buchanan $400
       Johnson   $200    $20000             Johnson  $200
Example i) The semi-join of
       CUST by AUTO_PAY on CUST.ChkBal=AUTO_PAY.Amount
    equals the join of
       CUST by AUTO_PAY on CUST.ChkBal=AUTO_PAY.Amount
      (Name,    ChkBal, LoanBal, Amount)
       Jeff.     $300    $30000   $300
       Johnson   $200    $20000   $200
    projected onto
      {Name,    ChkBal, LoanBal}
       Jeff.     $300    $30000
       Johnson   $200    $20000.
Example ii) The semi-join of
       CUST by AUTO_PAY on CUST.ChkBal<AUTO_PAY.Amount
    equals:
       Adams     $100    $20000
       Tyler     $100    $15000
    (These are customers whose balances are insufficient
    for their automatic loan payments.)
```

S onto the attributes of q (i.e. S.C), and moving the result to R's DM.

We define the <u>cost</u> of an operator to be the amount of inter-site communication it requires, and its <u>benefit</u> to be the amount by which it reduces its operand.* Under this definition, restriction and projection have zero cost and

---

*This definition is appropriate because communications is the bottleneck in a distributed DBMS.

non-negative benefit; hence they are always cost beneficial. Whether or not a semi-join is cost beneficial depends on the database state. The problem of Access Planning is to construct a program of cost beneficial semi-joins, given a transaction and a database state.

No procedure is known for producing optimum access plans in general (nor has any bound on the complexity of this problem been established). Instead heuristic methods are employed that find good, though not necessarily optimum, programs.

The procedure we employ uses a hill-climbing discipline, starting from an initial feasible program and iteratively improving it. The initial program is essentially the simple approach described at the beginning of this subsection. The Access Planner improves this program by first adding all restrictions and projections required by T, and then iteratively searching for cost-beneficial semi-joins. This procedure, like many hill-climbing algorithms, can be trapped by local optima and thereby fail to find the true optimum. While this problem is inherent in the approach, we alleviate it by using branch-and-bound techniques and other enhancements described in [WONG et al.].

2.1.4.2 Distributed Execution

The programs produced by the Access Planner are non-looping parallel programs and can be represented as data flow graphs [KARP and MILLER]. To execute the program, the TM issues commands to the DMs involved in each operation as soon as all predecessors of the operation are ready to produce output.

The effect of execution is to create at the final DM a temporary file to be written into the database (if T is an update) or displayed to the user (if T is a retrieval). At this point, the Execute phase has completed.

2.1.5 Reliable Writing

To complete transaction processing, the temporary file at the final DM must be installed in the permanent database and/or displayed to the user. For convenience let us say that the final DM has a set of temporary files $F_1,...,F_n$ to be installed at $DM_1,...,DM_n$ respectively; if any results must be displayed to the user, let us treat the user as one of the DMs. The problem is to ensure that failures cannot cause some DMs to install updates while causing others not to. We must protect against two types of failures: failure of a receiving DM, and failure of the sender; the former are handled by reliable delivery and the latter by transaction control, described in Sections 2.1.5.1 and 2.1.5.2 respectively.

Ideally one would like 100% protection against failures, but this goal is theoretically unattainable [GRAY]. Instead our goal is to attain acceptably high levels of protection, and, moreover, to make the level of protection a database design parameter.

2.1.5.1 Reliable Delivery

Reliable delivery guarantees that messages sent between pairs of sites are received in the order sent. Techniques for making this guarantee are well-known in the communication field as long as both sites are up. SDD-1 makes this guarantee even if the recipient is down when the message is sent, and the sender is down when the recipient recovers. Indeed, the two sites need never be up simultaneously.

Reliable delivery employs a mechanism called spoolers. A spooler is a process with unbounded memory (e.g., it has access to disk storage) that serves as a first-in-first-out message queue for a failed site. Any message sent to a failed site is delivered to its spooler instead. By employing multiple spoolers, arbitrarily high protection against multiple failures can be attained.

2.1.5.2 Transaction Control

Transaction control addresses failures of the final DM that occur during the Write phase. Suppose the final DM fails after sending files $F_1,\ldots,F_{k-1}$, but before sending $F_k,\ldots,F_n$. At this point, the database is inconsistent, because $DM_1,\ldots,DM_{k-1}$ reflect the effects of the transaction while $DM_k,\ldots,DM_n$ do not. Transaction control ensures that inconsistencies of this type are rectified in a timely fashion.

The basic technique employed is a variant of "two-phase commit" [GRAY]. During phase 1, the final DM transmits $F_1,\ldots,F_n$ but the receiving DMs do <u>not</u> install them yet. During phase 2, the final DM sends <u>Commit messages</u> to $DM_1,\ldots,DM_n$, whereupon each DM_i does the installation. If some DM, DM_k say, has received F_k, but not a Commit, and the final DM has failed, DM_k consults the other DMs. If any have received a Commit, DM_k does the installation; if none have received Commits, none do the installation, thereby aborting the transaction.

This technique offers complete protection against failures of the final DM, but is susceptible to multi-site failures. Enhancements that offer arbitrarily high protection against multiple failures are described in [HAMMER and SHIPMAN].

When updates are installed at all DMs the Write phase is completed. At this point the transaction has been fully processed.

2.1.6 Directory Management

SDD-1 maintains directories containing relation and fragment definitions, fragment locations, and usage statistics. Since TMs use directories for every transaction, efficient and flexible directory management is important. The main issues in directory management are whether or not to store directories redundantly, and whether directory updates should be centralized or decentralized. We have made these issues a matter of database design by treating directories as <u>ordinary user data</u>. This approach allows directories to be fragmented, distributed with arbitrary redundancy, and updated from arbitrary TMs.

But there are some problems. First, performance could be degraded by requiring that every directory access incur general transaction overhead, and by requiring that every access to remotely stored directories incur communication delays. We avoid these performance problems by caching recently referenced directory fragments at each TM, discarding them if rendered obsolete by directory updates. Since directories are relatively static, this solution is appropriate.

A second problem is that we now need a directory that tells where each directory fragment is stored. This directory is called the directory locator, and a copy of it is stored at every DM. This solution is appropriate because directory locators are relatively small, and extremely static.

With these enhancements, the SDD-1 directory management scheme combines the performance advantages of special-purpose directory management mechanisms with the flexibility of general-purpose data distribution and redundancy options.

2.1.7 Conclusion

SDD-1 is a general-purpose distributed DBMS, integrating database management, distributed processing, and reliable communication technologies into a cohesive system. This integration offers substantial benefits by combining the advantages of distributed processing with the advantages of centralized database management. At the same time it introduces new technical problems, of which the most critical are concurrency control, query processing, and reliable writing. This section has outlined the SDD-1 solutions to each of these problems; for in-depth presentations of our techniques we refer the reader to [BERNSTEIN et al. a,b] [BERNSTEIN and SHIPMAN a], [HAMMER and SHIPMAN], and [WONG et al.].

SDD-1 is the first general-purpose distributed DBMS ever developed. Its design was initiated in 1976 and completed in 1978. The first version of the system was released in 1978 and a complete prototype system will be released in 1979. SDD-1 is implemented for DEC-10 and DEC-20 computers running the TENEX and TOPS-20 operating systems; its communication medium is the ARPA network. SDD-1 is built on top of existing software to the extent possible; most notably it employs an existing DBMS, called Datacomputer [MARILL and STERN], to handle all database management issues. The current system is configured with four sites, although the software can support any reasonable number.

When we began the SDD-1 design, distributed database management was uncharted territory; now its major issues are known. SDD-1 has identified the major technologies upon which a distributed DBMS must be built, and the major new problems caused by integrating these technologies. The existence of SDD-1 as a system demonstrates that these problems can be solved in an integrated software system, and that distributed database management is indeed a feasible technology. The particular techniques we have developed for each new problem area are also significant and are among the best techniques known for each problem. In addition, much of our work has theoretical foundations extending beyond the SDD-1 context, and promises to form a strong framework for future research.

2.2 Concurrency Control

2.2.1 Literature Review

The concurrency control problem in database systems has been a major research focus for some time. In centralized database management systems (abbr. DBMSs), the conventional method to control concurrent update activity is two-phase locking [ESWARAN et al.]. Two phase locking requires that every transaction:

1. locks the data it reads and writes before it actually accesses it, and

2. does not obtain any new locks after it has released a lock.

Once a data item is locked, no other transaction may lock that data item until the owner of that lock releases it. Research into locking-based concurrency controls has analyzed deadlock problems, logical locks described by predicates (instead of by data item names), granularity of locks, and efficient locking algorithms [CHAMBERLIN et al.], [ESWARAN et al.], [GRAY et al.], [KING and COLLMEYER], [REIS and STONEBRAKER].

Locking methods have also been proposed for distributed DBMSs. One technique, called primary-site, uses a central lock controller to manage the locks [ALSBERG and DAY]. Alternatively, locks can be distributed with the data. Since data can be distributed redundantly, in principle all copies would have to be locked. To reduce locking overhead, one copy of each file (say) can be designated to be primary. Only the primary copy then needs to be locked, independent of which copies or how many copies are accessed [STONEBRAKER]. Variations of locking which set "imaginary locks" [THOMAS a,b] or which use version numbers [STEARNS et al.], [REED], [ROSENKRANTZ et al.] have also been proposed. (See also [BERNSTEIN and SHIPMAN b] for a proof that these methods are essentially locking approaches.)

These distributed locking approaches are quite similar to centralized concurrency controls, with the usual termination problems of indefinite postponement and/or deadlock. These mechanisms do differ, however, from centralized schemes in one respect -- the possibility of asynchronous failures of sites and communication links while an update is in the midst of being processed. Many of the proposed distributed concurrency controls have concentrated on this problem of failure (e.g., [ALSBERG and DAY] [MENASCE et al.] [STONEBRAKER] [THOMAS a,b]).

The concurrency control mechanism of SDD-1 differs from all of the above mechanisms in at least one way. In SDD-1, information about how transactions can conflict is preanalyzed before the transactions are submitted, so that not all transactions need synchronization. This preanalysis technique is the heart of the SDD-1 concurrency control and is the main topic of this section. Also, the run-time synchronization mechanisms of SDD-1, which differ considerably from locking, are discussed. An early restricted version of the SDD-1 concurrency control is discussed in [BERNSTEIN et al. a].

2.2.2 SDD-1 Transactions

The basic unit of user computation in SDD-1 is the transaction. The execution of each transaction is supervised by a TM and consists of three sequential steps:

1. The transaction reads a subset of the database, called its read-set, into a distributed private workspace.
2. It does some computation on the workspace.
3. The transaction writes some of the values in its workspace into a subset of the database, called its write-set. The write-set need not be included in the read-set.

Since the transaction is coded in terms of the logical database, and since the physical database in general has redundant copies of many logical data items, the TM must choose which physical copies of the logical data items referenced by the transaction should be read or written. To keep the physical database internally consistent, the TM must apply each write operation on a data item to all physical copies of that data item. However, only one of the physical copies of each logical data item needs to be used for reading.

To obtain the read-set data for a transaction's input and later to write its output into copies of its write-set, a TM sends READ and WRITE messages to DMs. A READ message is a request by a TM to read some of the data items stored at a DM and to store them in a local workspace at that DM on behalf of some transaction. A WRITE message is sent by a TM to a DM to report updates produced by a transaction which the TM supervised.

To process a transaction, a TM must send READ messages to obtain the transaction's read-set. Logical data items are obtained from physical copies selected by the TM. The TM sends a READ message to those DMs that store the selected copies to be read by the transaction.

After all READ messages have been processed (i.e., after they have been positively acknowledged), the TM supervises the execution of the transaction. This function of the TM is performed by the access planner and is described in [WONG et al]. It is the job of the concurrency control mechanism to guarantee that the physical read-set obtained by READ messages is internally consistent, so that the transaction will produce correct output.

Write operations performed by the transaction are put into a temporary file and are deferred until the transaction completes execution. After the transaction completes execution, the TM broadcasts these updates to DMs as WRITE messages. Each update to a logical data item, say x, is sent to <u>all</u> DMs that have a stored copy of x.

A TM sends at most one READ message and at most one WRITE message to each DM on behalf of a single transaction. If, for example, a transaction reads data from two data items that reside at the same DM, then only one READ message is issued to read both data items. This is an important point, as each DM performs READs and WRITEs as atomic operations; for example, none of the data read by a READ message can be updated by some WRITE while the READ is being processed.

2.2.3 Concurrent Correctness

The system usually has many transactions in progress at any one time, both because there are multiple TMs operating concurrently within the system and because individual TMs are processing transactions concurrently. If the READs and WRITEs that implement these transactions were arbitrarily interleaved, then serious problems of database consistency would result. The usual method of avoiding these consistency problems is by guaranteeing that the execution of transactions is serializable [ESWARAN et al] [PAPADIMITRIOU et al] [ROSENKRANTZ et al].

We say that an interleaved execution of a set of transactions is <u>serializable</u> if it is "equivalent" to a history of operation in which each of the transactions runs alone to completion before the next one begins. Two executions are <u>equivalent</u> if in both executions each transaction produces the same output, thereby leading to the same final state of the database. That is, an interleaved execution is serializable if it could be reproduced by a non-interleaved (i.e., serial) execution of the same set of transactions. Note that serializability requires only that there exists <u>some</u> serial order equivalent to the actual interleaved execution. There may in fact be <u>several</u> such equivalent serial orderings.

The adoption of serializability as the criterion for concurrent correctness is based on the assumption that each user transaction will preserve database consistency if it runs atomically. That is, if only one transaction is allowed to execute at a time, and if the database state is initially consistent, then after executing a transaction the database state must still be consistent. So, a serial ordering of transaction executions will, by induction, result in a consistent database state. Since a serializable execution is equivalent to some serial one, a serializable execution results in a consistent database state as well.

The issue of serializability arises because a system's atomic actions are at a finer granularity than its users' atomic actions. In SDD-1, the users' atomic actions are transactions, while the system's atomic actions are the execution of READ and WRITE messages at the DMs. Each DM behaves as if READs and WRITEs are processed atomically, so it is impossible for a READ operation to observe the effects of only a part of a WRITE operation at a DM.

When a system allows the execution of several transactions at the same time, then the system's operations corresponding to different transactions are interleaved. If the interleaving is not controlled, there is no guarantee that the behavior of such a system conforms to the user's expectation that each transaction is processed as an indivisible computation.

For example, assume there is a single copy of data item x, which initially has the value x=0. There are two transactions in the system; transaction i sets x:=x+1, and transaction j sets x:=x+2. The following sequence of events occurs:

Transaction i reads x=0

Transaction j reads x=0

Transaction j sets x:=2

Transaction i sets x:=1

Any serial execution of the two transactions, one after the other, would have resulted in setting x to 3. However, the result of this interleaved execution is to set x to 1, contrary to the user's intention. This execution history is not serializable, since no serial processing of these transactions will produce the observed effects.

To guarantee serializability in SDD-1, we apparently need to avoid undesirable interleavings of READ and WRITE messages -- those that lead to nonserializable executions. We accomplish this goal using two mechanisms. First, we examine each transaction to determine if it is conceivable that it could participate in a nonserializable execution. As we will see, many transactions will never produce READs and WRITEs that interleave badly with other transactions, and hence can be run unsynchronized. Second, for those transactions that are determined to be dangerous because they can participate in nonserializable executions, we synchronize their READ and WRITE messages using protocols that avoid undesirable interleavings. These protocols are based on a timestamping mechanism and are quite different from the locking protocols used in conventional centralized DBMSs.

As we will see, most of the effort in distinguishing transactions that require no synchronization from the dangerous ones is done statically when the database is designed. When a transaction is actually submitted, a simple local table look-up is sufficient to determine how much, if any, synchronization is required. The run-time mechanism is the collection of protocols that must be invoked for those transactions that do require synchronization.

Note that these two components of the concurrency control mechanism are independent. Our technique for analyzing transactions to determine sources of nonserializability could be used in conjunction with conventional locking protocols. Or, we could run <u>all</u> transactions using our timestamp-based protocols and ignore the preanalysis step entirely, as in present systems that use locking without preanalysis. Together the two mechanisms provide a powerful technique for synchronizing concurrent transactions at low cost.

Before describing the heart of the system -- the method for determining the amount of synchronization required by each transaction and the protocols that cffect that synchronization -- we must first describe two basic concepts that underlie much of the concurrency control mechanism. These concepts, timestamps and transaction classes, are described in the next two sections.

2.2.4 Timestamps

Each transaction executed by SDD-1 is assigned a globally unique timestamp. Transaction timestamps serve a number of purposes for synchronizing READs and WRITEs. To generate globally unique timestamps, a TM reads its local clock and appends its unique TM number as the low order bits of the timestamp. By requiring that once a clock is read it cannot be read again until it has been incremented, we ensure that every timestamp is globally unique within the system [THOMAS a].

The clocks are actually maintained as part of the Reliable Network, the reliable communications facility of SDD-1. By using the clock synchronization method described in [LAMPORT], the system behaves as if there were a single virtual clock available to all sites.

One use of timestamps is in processing WRITE messages that arrive at a DM out of order. The problem is that the WRITE messages sent by two transactions that update the same logical data item may be processed in different orders at different DMs, thereby producing mutually inconsistent copies of the data item. One way to solve this problem is to attach the transaction's timestamp to all of its WRITE messages, and then require that WRITE messages be processed in timestamp order at all DMs. A better method that gives more flexibility to DMs in the processing of WRITE messages uses timestamped data items and is adopted in SDD-1 (this method was originally suggested in [THOMAS a]).

A transaction's timestamp is carried on all of its WRITE messages. In addition, every physical data item at every DM has an *associated timestamp.* Note that timestamps are attached to <u>physical</u> data items; there may be many physical copies of a logical data item and each one has its own attached timestamp. The timestamp of a data item is the timestamp of the last WRITE message that updated it. Each DM processes WRITE messages according to the following <u>WRITE message rule</u>: A data item is updated by a WRITE message if and only if the data item's timestamp is less than the WRITE message's timestamp. (Recall that a WRITE message contains the final values of data items, not computations to be performed on them.) So, to process a data item in a WRITE message, the DM compares the timestamp of the WRITE message with the timestamp of its stored copy of the data item. If the timestamp of the WRITE message exceeds the timestamp of the stored data item, then the new value of the data item in the WRITE message is written into the stored data item along with the new timestamp. Otherwise, the update is not performed <u>on that stored data item</u>. This is a data item by data item check; some data items in the WRITE message may result in update operations while others may not.

Whenever a WRITE message for a recent transaction that updates some data item is processed at a DM before a WRITE message for an earlier (i.e., older) transaction that updates the same data item, the latter WRITE message will contain a data item update that is not performed. Such a situation is not an error. It is simply the way that the system reorders updates to occur in the same order that their generating transactions executed. That is, the net effect of a set of WRITE messages processed at a DM in arbitrary order is the same as the effect of processing them in timestamp order without the WRITE message rule. The principal advantage of using the WRITE message rule is that WRITE messages can be processed as soon as they are received, thereby avoiding artificial queuing delays at the DMs.

Note that the correctness of the WRITE message rule in reordering updates does not require that clocks in different TMs be at all synchronized. This is true of other timestamp related mechanisms in SDD-1 as well. For reasons of efficiency, however, it is necessary to assume that clock values in different TMs are reasonably close to each other.

A principal objection to timestamped data items is its cost. However, not all timestamps actually need to be stored. If the timestamp of a data item is earlier than the timestamp of any transaction whose WRITE messages have not yet been processed, then the data item's timestamp is effectively zero. Any WRITE message that tries to update that data item will succeed, because the WRITE message will have a later timestamp than the data item. So, we need only maintain the timestamps of recently updated data items. If a data item is not updated for a while (say a few minutes), then its timestamp can be assumed to be zero and therefore dropped. A caching mechanism for timestamps using differential files is used in SDD-1 for this purpose. Using this mechanism, we judge that the overhead in maintaining timestamps will be small, since only a small portion of the data items will require their timestamps to be stored in the cache at one time.

2.2.5 Transaction Classes

A crucial aspect of the SDD-1 concurrency control mechanism is its ability to distinguish between transactions that require synchronization and those that do not. By examining the read-set and write-set of transactions, the system can determine which transactions conflict with each other. Intuitively, two transactions _conflict_ if the read-set or write-set of one intersects the write-set of the other. Such conflicts are the main cause of nonserializability. They are avoided in conventional DBMSs by locking data items so that two conflicting transactions never run concurrently. However, preventing _all_ conflicts is more than what is required to guarantee serializability. By analyzing a graph theoretic representation of the transactions, called a _conflict graph_, the system can isolate the dangerous conflicts that can potentially lead to nonserializability. This analysis technique will be described in detail later in the report.

Unfortunately, analyzing the conflict graph at run-time for all executing transactions is too time consuming. Also, since the transactions are distributed at run-time, assembling a conflict graph would require too much communication. So, we transform this run-time analysis into a static analysis done only once at database design time by capitalizing on the predictability of transaction types in the following way.

When designing the database, the database administrator establishes a static set of transaction classes. Formally, each transaction class is defined by a logical read-set and write-set and is assigned to run at a particular TM. A transaction fits in a class if the read-set and write-set of the transaction is contained (respectively) in the read-set and write-set of the class. Read-set and write-set definitions are expressed using simple predicates, so that class membership can be checked quickly (see Figure 2.9).

The conflict graph analysis is now done on the statically defined transaction classes instead of on the transactions themselves. This analysis yields the type of synchronization, if any, required for each class. At run-time, when a transaction is submitted to a TM, the TM selects a class in which the transaction fits and applies the type of synchronization specified by the analysis for that class.

Class Definitions Using Simple Predicates   Figure 2.9

```
 Relation Schema:  INVENTORY (ITEM#,DESCRIPTION,PRICE,
                                 QUANTITY)

Class 1
   read-set:   INVENTORY [ITEM#,PRICE]
   write-set:   INVENTORY [PRICE]
   comments:   transactions that update prices

Class 2
   read-set:   INVENTORY [ITEM#,QUANTITY]
                     WHERE (PRICE > $100)
   write-set:   INVENTORY [QUANTITY]
   comments:   transactions that update quantities of
               high-priced items

Class 3
   read-set:   INVENTORY [ITEM#,DESCRIPTION,PRICE]
                           WHERE (QUANTITY > 0)
   write-set:   user's terminal
   comments:   transactions that display item information
               about items currently in stock.
```

The utility of classes lies in the property that two transactions that run in different classes conflict only if their classes conflict. Hence, conflicts between transactions can be determined by conflicts between classes. So, an analysis of the classes at database design time is sufficient to determine potentially dangerous conflicts between transactions at run time. We believe that, for many kinds of applications, the most frequent determination will be that the class participates in no dangerous conflicts and can therefore run with only local synchronization.

How a TM Processes a transaction Figure 2.10

```
Do Forever;
   Wait for a transaction, T, to arrive;
   Find a class, C, in which T fits;
   If C cannot be processed locally
    then forward T to a site that can process C
    else begin
         look up the synchronization rules for class C,
         send out appropriate READ messages on
             behalf of T, synchronizing where necessary;
         supervise the distributed execution of T;
         send out WRITE messages on behalf of T
         end
   end
```

For a set of class definitions to be feasible, it must cover _all_ transactions that might ever be submitted. It is _not_ necessary that _every_ TM have enough classes to accept all possible transactions, since a TM can forward a transaction to some other TM for execution. However, it _is_ necessary that every possible transaction fit in a class supported by _some_ TM. A sketch of how a transaction is routed and executed by TMs appears in figure 2.10.

2.2.6 Synchronizing Transactions Within a Class

To ensure the serializability of transactions which execute in the same class, we require that within a class all of the transactions are actually executed serially, one after another. To formalize this requirement, some notation is helpful. Let the processing of a READ message on behalf of transaction i at DM_{alpha} * be denoted R^{i}_{alpha}. Similarly, let the processing of a WRITE message on behalf of transaction i at DM_{alpha} be denoted W^{i}_{alpha}. Then we can express the requirement that transactions within a class run serially as follows:

<u>Class Pipelining Rules</u>: For each DM_{alpha}, for each class $\bar{I}$, and for each pair of transactions i_1 and i_2 in $\bar{I}$,

C1. If i_1 and i_2 both read from DM_{alpha}, then $R^{i_1}_{alpha}$ is processed before $R^{i_2}_{alpha}$ only if i_1 has an earlier timestamp than i_2.

---

* We use lower case Greek letters to denoted DMs. We use lower case Roman letters i,j,k,... to denote transactions. We denote the class in which transaction i executes by $\bar{I}$.

C2. If i_1 and i_2 both write into DM_{alpha}, then $W^{i_1}_{alpha}$ is processed before $W^{i_2}_{alpha}$ only if i_1 has an earlier timestamp than i_2.

C3. If i_1 reads some data item at DM_{alpha} and i_2 writes some data at DM_{alpha}, then $R^{i_1}_{alpha}$ is processed before $W^{i_2}_{alpha}$ only if i_1 has an earlier timestamp than i_2.

The class pipelining rules force transactions that run in a single class to be processed serially at all DMs in the same order. Rules C1 and C2 guarantee that READ and WRITE messages (respectively) from each class are processed in timestamp order at all DMs. Rule C3 guarantees that READ messages from each class only see updates from earlier WRITE messages in that class.* These rules are sufficient to guarantee the noninterference of any two transactions that run in a single class.

The class pipelining rule, although stated in terms of DMs, is actually enforced by mechanisms at both TMs and DMs. For each class that a TM processes, the messages from that class are sent to each DM in an order that is

---

* Actually, a weaker condition than C3 is possible. C3 must only be applied when the read-set of i_1 at DM_{alpha} intersects the write-set of i_2 at DM_{alpha}, since this is the only case when i_1 can actually see the update produced by i_2. However, to eliminate several special analyses that would be required, we assume C3 is always applied.

consistent with C1-C3. The communications network (ARPANET, in our case) guarantees that messages are received in the order they were sent, for any point-to-point communications channel. The DMs process messages within a class in the order in which they are received, thereby enforcing C1-C3.

2.2.7 Interclass Interference

2.2.7.1 An Example of Safe Interference

We say that a set of transactions <u>interfere</u> if the system allows them to be interleaved in a nonserial manner. Given the class pipelining rule, we need not be concerned with interference among transactions in the same class, since they are run serially. The problem now is to avoid interference among transactions in different classes. A critical aspect of our solution to this problem is isolating those cases where transactions in different classes never interfere with each other. This requires some subtlety, for even when transactions read and write the same data items, they may not interfere, as illustrated by the following simple example.

Suppose we run two transactions, say i and j, in two different classes, $\bar{I}$ and $\bar{J}$, each of which first finds the EMPLOYEE record whose NAME domain has the value 'JON DOE'; then each writes a distinct new value into the PHONE# domain of that record (the phone numbers written by the two transactions are different). Naturally, the final value of JON DOE's PHONE#, after both transactions execute, is dependent on the order in which their write operations were processed. However, no matter how their read and write operations are interleaved, the execution will be serializable. The transactions will always appear to have executed serially with the order of their writes determining the order of the transactions in the serialization; the transaction that writes JON DOE's PHONE# first appears first in the serialization. Therefore, even though the transactions have overlapping write-sets -- a situation that conventionally requires locking -- no synchronization is necessary.

To exploit situations of this type, we must determine safe patterns of interleaved reads and writes that require no synchronization. This determination is accomplished by analyzing conflicts between transaction classes. For example, an analysis of classes $\bar{I}$ and $\bar{J}$ above would show that all patterns of interleaved reads and writes are serializable. This analysis is performed on a graph theoretic representation of transaction conflicts, and is the subject of the next section.

2.2.7.2 Conflict Graphs

As we observed in Section 2.2.5, two transactions from different classes conflict only if their classes conflict.

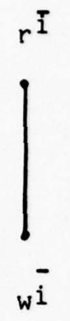
To formalize this, we say that WRITE message W^{i}_{alpha} <u>conflicts with</u> a READ message R^{j}_{alpha} iff transaction i's write-set intersects transaction j's read-set. A WRITE message W^{i}_{alpha} <u>conflicts with</u> another WRITE message W^{j}_{alpha} iff transaction i's write-set intersects transaction j's write-set. It follows that if R^{i}_{alpha} conflicts with W^{j}_{alpha}, then the read-set of class $\overline{i}$ intersects the write-set of class $\overline{j}$. By examining class conflicts, we can predict potential transaction conflicts, which are a primary component of the serializability problem. It will turn out that this examination of class conflict will lead us to our goal -- a method for determining the amount of synchronization required by each transaction.

The method begins with the construction of a <u>conflict graph</u> (see Figure 2.11). In the graph, each class, say $\overline{i}$, is modeled by two nodes labelled $r^{\overline{i}}$ and $w^{\overline{i}}$. For each class, $\bar{i}$, an edge $<r^{\bar{i}},w^{\bar{i}}>$ connecting them is drawn (Figure 2.11a). When the write-sets of two classes, say $\bar{i}$ and $\bar{j}$, intersect, then the edge $<w^{\bar{i}},w^{\bar{j}}>$, called a <u>horizontal edge</u>, is drawn (Figure 2.11b). Similarly, if the read-set of one class (say $\bar{i}$), intersects the write-set of another class (say $\bar{j}$), then an edge $<r^{\bar{i}},w^{\bar{j}}>$ called a <u>diagonal edge</u> is drawn (Figure 2.11c).

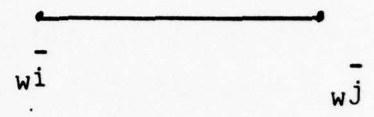
For a given set of classes, $\underline{C}$, we denote the conflict graph for $\underline{C}$ by $CG_{\underline{C}}$. A sample conflict graph appears in Figure 2.12.

We will use the conflict graph to help us predict the amount of synchronization required by each transaction class. The connection between synchronization protocols and conflict graphs is developed in Section 2.2.8. Since this development is lengthy and may not be of interest to all readers, we summarize the principle results of Section 2.2.8 in Section 2.2.9. Hence Section 2.2.8 can be skipped, if desired, without loss of continuity.

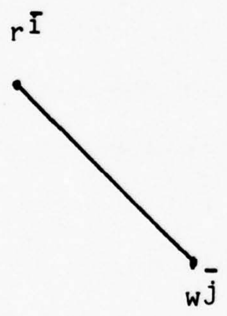
Conflict Graph Edges Figure 2.11



(a) a vertical edge is drawn between every $r^{\bar{i}}, w^{\bar{i}}$ pair.

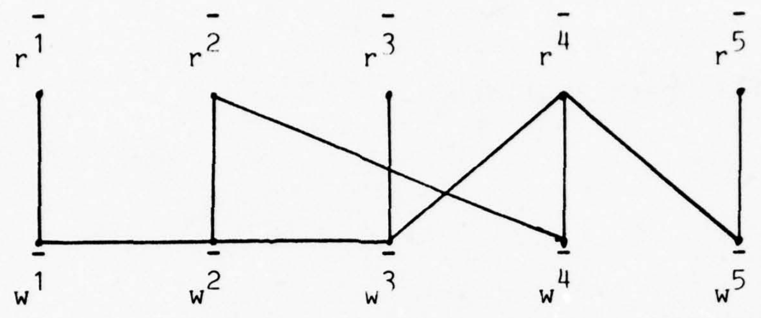


(b) a horizontal edge is drawn between a $w^{\bar{i}}, w^{\bar{j}}$ pair iff the write-sets of $\bar{i}$ and $\bar{j}$ intersect.



(c) a diagonal edge is drawn between an $r^{\bar{i}}, w^{\bar{j}}$ pair iff the read-set of $\bar{i}$ intersects the write-set of $\bar{j}$.

A Sample Conflict Graph Figure 2.12



2.2.8 Conflict Graph Analysis

2.2.8.1 Serializing Logs

Depending on the order in which READ and WRITE messages are processed by the system, an interleaved execution of transactions may or may not be serializable. To understand which message orderings _are_ serializable, we need a notation that models these orderings. In our notation, we will represent the ordered processing of READ and WRITE messages at a DM by a log. A _log_ is simply a string of R's and W's that have the same DM subscript. For example, R^1_{alpha} W^2_{alpha} W^1_{alpha} R^5_{alpha} W^5_{alpha} R^4_{alpha} is a log describing the order in which READ and WRITE messages were processed at DM_{alpha}. When we say, for example, that R^i_{alpha} _precedes_ W^j_{alpha} (in DM_{alpha}'s log), we mean that R^i_{alpha} _was processed before_ W^j_{alpha} at DM_{alpha}.

A log is a complete representation of the computations performed on the database at a DM. If we were given the list of data items read and written by each WRITE message as well as the timestamps of transactions (so that we could correctly apply the WRITE message rule), then we would be able to reproduce the computation that was actually performed at the DM. So, an "interleaved execution of transactions" in SDD-1 is modelled by a "collection of DM logs, one per DM". We will therefore use these two terms interchangeably.

Suppose we are given an interleaved execution of N transactions, represented by a set of DM logs. Which of the N! possible serializations of the transactions is an equivalent serialization of the given logs? A serialization is <u>equivalent</u> to the given logs if that serial execution of the transactions on a nondistributed, nonredundant database (represented by the serialization) produces the same computation as the interleaved execution on the distributed, redundant database (represented by the DM logs). It is a theorem that <u>if each transaction reads from a database that has had exactly the same write operations applied to it in the serialization as were applied to it in the given interleaved execution, then each transaction will perform the same computation in the serialization as it did in the given interleaved execution</u> [PAPADIMITRIOU et al]. We can guarantee this condition by requiring that the serialization satisfy the following three rules: For each i,j, and alpha

1. If W^{i}_{alpha} precedes and conflicts with R^{j}_{alpha}, then i must precede j in the serialization;
2. If R^{j}_{alpha} precedes and conflicts with W^{i}_{alpha}, then j must precede i in the serialization;
3. If W^{i}_{alpha} conflicts with W^{j}_{alpha}, then i and j must appear in the serialization in their timestamp order.

If the serialization obeys (1) and (2), then write operations in the serialization precede exactly the same read operations as they did in the given interleaved execution. However, this is not the same as saying that each transaction reads from a database that has had exactly the same write operations applied to it in the serialization as were applied to it in the given execution. The reason is that due to the WRITE message rule, the order in which WRITE messages are processed is not the same as the effective order in which the write operations are applied to the database; indeed, some write operations are not applied at all. To understand this subtle distinction is to understand the need for rule (3).

In the logs, the WRITE message rule prevents certain write operations from being applied; this occurs when a WRITE message with an early timestamp arrives after a WRITE message with a later timestamp and both WRITE messages write into a common data item. The WRITE message rule is an artifact of the distributed execution of SDD-1, and would not have been applied if the transaction were executed serially on a nondistributed, nonredundant database. In essence, this means that the serialization must produce the same computation <u>without</u> the WRITE message rule that the given logs produced <u>with</u> the WRITE message rule. Rules (1) and (2) alone are not strong enough to make this guarantee.

For example, suppose the log for DM_{alpha} contains W^{i}_{alpha} W^{j}_{alpha} R^{k}_{alpha} where j has an <u>earlier</u> timestamp than i and all three messages write into or read from data item x. The WRITE message rule prevents W^{j}_{alpha} from overwriting x, so R^{k}_{alpha} reads x from W^{i}_{alpha}. We want the same relative ordering of R^{k}_{alpha} and W^{i}_{alpha} to appear in the serialization. So, transaction j must precede transaction i in the serialization. However, the serialization [i,j,k] would be permitted by the rules (1) and (2) alone; this is incorrect because transaction k would read x from j (not i) in this serialization.

Rule (3) guarantees that write operations in the serialization are applied in the same relative order as they are applied in the given logs. It "factors out" the WRITE message rule from the serialization by requiring the write operations to appear in the order that they were effectively applied, rather than the order in which they were processed.

By developing rules (1) - (3), we have related the order of conflicting READ and WRITE messages in DM logs to the order of transactions in serializations. As we know, not all interleaved executions are serializable. So, as we would expect, there are DM logs that have no serialization that obeys rules (1)-(3). In principle, we could schedule READ and WRITE messages by continually checking rules (1)-(3) at run-time so that the order in which READ and WRITE messages are processed can always be serialized. However, this would be very costly in computation time and communication traffic. Instead, we use the conflict graph model of transaction conflicts to guide us in synchronizing READ and WRITE messages so that a serialization obeying rules (1)-(3) is always possible.

The conflict graph is used to determine potentially nonserializable executions of conflicting transactions. The interpretation of diagonal and horizontal edges can be used to extend rules (1) - (3): For each i, j, and alpha

1'. If $<\bar{w}^{i},\bar{r}^{j}>$ is a diagonal edge of CG and W^{i}_{alpha} precedes R^{j}_{alpha} in DM_{alpha}'s log, then i must precede j in any serialization.

2'. If $<\bar{r}^{i},\bar{w}^{j}>$ is a diagonal edge of CG and R^{i}_{alpha} precedes W^{j}_{alpha} in DM_{alpha}'s log, then i must precede j in any serialization.

3'. If $<\bar{w}^{i},\bar{w}^{j}>$ is a horizontal edge of CG, then i and j must appear in the serialization in their timestamp order.

Since two transactions conflict only if their classes conflict, any serialization that satisfies (1')-(3') will satisfy (1)-(3) as well. The advantage to using (1') - (3') in place of (1) - (3) is that the former are stated entirely in terms of class conflicts, which are known in advance.

In SDD-1, there is always a serialization of the executed transactions that satisfies (1')-(3'). The mechanisms that are used to guarantee that such a serialization always exists are called <u>protocols</u>.

2.2.8.2 Protocol P1 and the Acyclicity Theorem

To understand why we need protocols, let us consider a system consisting of two classes, say $\bar{i}$ and $\bar{j}$, such that only one transaction is processed in each class, say transactions i and j. Under what conditions will these two transactions be serializable? If there are no horizontal or diagonal edges connecting $\bar{i}$ and $\bar{j}$ in the conflict graph, then (1')-(3') are trivially satisfied. In this case, i and j are serializable; in fact, either serialization will do. What if $\bar{i}$ and $\bar{j}$ are connected by some edge?

If $<w^{\bar{i}},w^{\bar{j}}>$ appears in CG, and if W^{i}_{alpha} and W^{j}_{alpha} are processed (for some DM_{alpha}), then according to rule (3') i and j must be serialized in timestamp order. If this is the only edge connecting $\bar{i}$ and $\bar{j}$, then the transactions are still surely serializable. For no matter how many DMs process WRITE messages from both transactions, each DM will apply the WRITE message rule, thereby making it look like i was processed before j. Therefore, applying rule (3') at all DMs will yield the same requirement that i and j be serialized in the same timestamp order. The only way we could get into trouble is if one DM believes i should precede j in the serialization while another believes j should precede i -- a clear impossibility using (3'). So, if $\langle w^{\bar{i}}, w^{\bar{j}} \rangle$ is the only edge connecting $\bar{i}$ and $\bar{j}$, we are safe.

If $\langle r^{\bar{i}}, w^{\bar{j}} \rangle$ appears in CG, then we have a potential problem. Suppose W^j_{alpha} precedes and conflicts with R^i_{alpha} and R^i_{beta} precedes and conflicts with W^j_{beta}. Rule (1') applied at DM_{alpha} says that j should precede i while rule (2') applied at beta says that i should precede j. Since both cannot be simultaneously satisfied, we have a nonserializable interleaving. Apparently, we must introduce some synchronization mechanism to avoid this problem produced by the diagonal edge.

Protocol P1 is the mechanism used to synchronize diagonal edge conflicts. We say that <u>transaction i obeys protocol P1 with respect to transaction j</u> if the relative ordering of READ messages from i and WRITE messages from j are the same at all DMs where both appear and conflict. Stated more formally, if R^i_{alpha} precedes (resp. follows) and conflicts with W^j_{alpha} at DM_{alpha}, then if R^i_{beta} and W^j_{beta} conflict and both are processed at DM_{beta}, R^i_{beta} must precede (resp. follow) W^j_{beta} at DM_{beta}.

We require that if $\langle r^{\bar{i}}, w^{\bar{j}} \rangle$ is an edge in CG, then for each pair of transactions i in class $\bar{i}$ and j in class $\bar{j}$, i must obey protocol P1 with respect to j. If the protocol is obeyed, then the nonserializable situation due to the opposite serializations implied by rules (1') and (3') cannot occur. Since this is the only problem a single diagonal edge can cause, P1 is sufficient to synchronize diagonal edges.

The above observations regarding single edge conflicts between two classes generalize directly to paths of conflicts. Suppose there is a single edge conflict between $\bar{i}$ and $\bar{k}$, and another one between $\bar{k}$ and $\bar{j}$. Again, assume one transaction runs in each class, say i, j, and k. Rules (1')-(3') only restrict the order of serialization between pairs of conflicting transactions. They will either require that i and j have a defined relative ordering (i.e., either i precedes k and k precedes j or i follows k and k follows j) or that they have no special required order (i.e., either i precedes k and j precedes k or i follows k and j follows k). In either case, the three transactions are serializable.

The only way the transactions might not be serializable is if there were two different paths from $\bar{i}$ to $\bar{j}$. Then, one path could lead to i preceding j according to rules

(1')-(3'), while the other path could lead to i following j. If this occurred, then the execution would be nonserializable. But note that it can only occur if there are <u>two</u> distinct paths. Two distinct paths that link $\bar{I}$ to $\bar{J}$ constitute a cycle. So, as long as there are no cycles in the conflict graph and each class runs one transaction, P1 is sufficient to guarantee serializability.

The class pipelining rule requires that transactions within a single class essentially run serially. So, the above statement about acyclic conflict graphs generalizes to the case of multiple transactions per class. (A proof of this fact is nontrivial and appears in [BERNSTEIN and SHIPMAN a].)

Our observations in this section can now be stated more formally as follows:

<u>Acyclicity Theorem</u> For a given set of transaction classes, $\underline{C}$, if

1. $CG_{\underline{C}}$ has no cycles, and
2. all classes in $\underline{C}$ obey the class pipelining rule, and
3. for each diagonal edge $\langle r^{\bar{I}}, w^{\bar{J}} \rangle$ in $CG_{\underline{C}}$ and transactions i in $\bar{I}$ and j in $\bar{J}$, transaction i obeys P1 with respect to j,

then all possible interleavings of transactions in classes in $\underline{C}$ are serializable.

To make the acyclicity theorem effective, we need to demonstrate an implementation for P1. This we will do in Section 2.2.10. First, however, we will show how to synchronize nonserializable situations caused by cycles.

2.2.8.3 Cycles, P3, and the Serializability Theorem

We have shown that if no cycles exist in the conflict graph and if P1 is properly applied, then all possible interleaved executions of transactions will be serializable. We also observed that cycles in the conflict graph can cause a nonserializable execution. If two distinct paths exist between two classes, $\bar{i}$ and $\bar{j}$, then the paths may lead to opposite serializations of transactions i in $\bar{i}$ and j in $\bar{j}$ according to rules (1')-(3') -- a nonserializable situation. To eliminate this possibility, we introduce a protocol that forces any two paths between $\bar{i}$ and $\bar{j}$ to always lead to the same relative ordering of i and j in all serializations. To illustrate the problem and the protocol that solves it, let us consider another example.

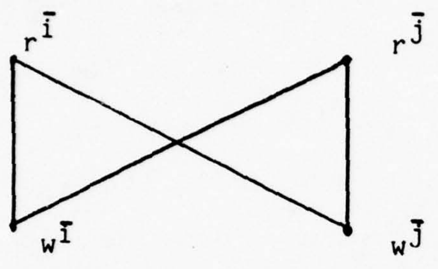
This time, suppose the database has one data item, x, stored at DM_{alpha}. Classes $\bar{i}$ and $\bar{j}$ both read from and write into x; for example, they both run transactions that increment x. The conflict graph for these classes contains two distinct edges, $\langle r^{\bar{i}},w^{\bar{j}}\rangle$ and $\langle w^{\bar{i}},r^{\bar{j}}\rangle$, connecting $\bar{i}$ and $\bar{j}$. These two edges together with $\langle r^{\bar{i}},w^{\bar{i}}\rangle$ and $\langle r^{\bar{j}},w^{\bar{j}}\rangle$ constitute a cycle (see Figure 2.12.2). The problem is that the diagonal edges may force opposite serializations of transactions in $\bar{i}$ and $\bar{j}$.

Consider, for instance, transactions i in $\bar{i}$ and j in $\bar{j}$ which execute their READ and WRITE messages in the following order: R^{i}_{alpha} R^{j}_{alpha} W^{i}_{alpha} W^{j}_{alpha}. Notice that P1 is trivially obeyed since there is only one DM. Since R^{i}_{alpha} precedes and conflicts with W^{j}_{alpha}, rule (2') implies that i must be serialized before j. Since R^{j}_{alpha} precedes and conflicts with W^{i}_{alpha}, the same rule implies that j must be serialized before i. Since both cannot be simultaneously satisfied, the execution is nonserializable. This occurred because the edges between $\bar{i}$ and $\bar{j}$ led to opposite serializations.

Protocol P3 prevents executions such as this one by making the following guarantee: <u>If two transactions belong to two classes connected by a diagonal edge in a cycle, then the timestamp order of the two transactions is the same as</u>

A Conflict Graph Cycle and Nonserializable Execution

Figure 2.13



Classes $\bar{I}$ and $\bar{J}$ have data item x in their read-sets and write-sets.

(a) The Conflict Graph

log for DM_{alpha}: R^{i}_{alpha} R^{j}_{alpha} W^{i}_{alpha} W^{j}_{alpha}

(b) A nonserializable log of transactions from class $\bar{I}$ and $\bar{J}$.

<u>the relative ordering dictated by rules (1') or (2') applied to the messages that correspond to the edge</u>. Before examining how P3 accomplishes this task, let us first see how P3 corrects the above example.

Since $[\langle r^{\bar{I}}, w^{\bar{J}}\rangle, \langle w^{\bar{J}}, r^{\bar{J}}\rangle, \langle r^{\bar{J}}, w^{\bar{I}}\rangle, \langle w^{\bar{I}}, r^{\bar{I}}\rangle]$ comprises a cycle, P3 applies to transactions i and j. Suppose that the timestamp of i is smaller than the timestamp of j. We observed that rule (2') required that i be serialized before j because R^i_{alpha} precedes W^j_{alpha}, and that j be serialized before i because R^j_{alpha} precedes $\underline{W}^i_{al\underline{p}ha}$. But the latter requirement violates P3. Since $<r^j, w^i>$ is in a cycle, protocol P3 implies that rule (2') applied to R^j_{alpha} and W^i_{alpha} must lead to i and j being serialized in timestamp order. However, the opposite occurred. What P3 must do, therefore, is make sure that W^i_{alpha} precedes R^j_{alpha}. Then both edges will lead to i and j being serialized in timestamp order and the nonserializability problem goes away.

Formally, we define protocol P3 as follows. <u>A transaction i *obeys protocol* P3 with respect to transaction j at DM_{alpha}</u> if R^i_{alpha} and W^j_{alpha} are processed in timestamp order. We require that for each diagonal edge $<r^{\bar{i}}, w^{\bar{j}}>$ in a cycle and for each i, j and alpha such that R^i_{alpha} conflicts with W^j_{alpha}, i must obey P3 with respect to j at DM_{alpha}.

Protocol P3 synchronizes multi-class cycles as well as the simple two-class cycle just illustrated. In a cycle consisting of several diagonal and horizontal edges, P3 requires that each conflict due to a diagonal edge leads to the pair of transactions being serialized in timestamp order. Rule (3') makes the very same requirement for horizontal edges. So, insofar as this cycle is concerned, if rules (1')-(3') say anything about the relative ordering of two transactions whose classes are on the cycle, then the requirement must be that the transactions be serialized in timestamp order. Since there is only one timestamp ordering of transactions, conflicting serialization orderings are impossible. Generalizing this observation for the case of multiple transactions per class as we did for the acyclicity theorem leads to the correctness theorem for the SDD-1 concurrency control.

Serializability Theorem For a given set of transaction classes, $\underline{C}$, if

1. all classes in $\underline{C}$ obey the class pipelining rule, and
2. for each diagonal edge $\langle r^{\bar{i}}, w^{\bar{j}} \rangle$ in $CG_{\underline{C}}$ and transaction i in $\bar{i}$ and j in $\bar{j}$, transaction i obeys P1 with respect to transaction j, and
3. for each diagonal edge $\langle r^{\bar{i}}, w^{\bar{j}} \rangle$ in a cycle in $CG_{\underline{C}}$ and transaction i in $\bar{i}$ and j in $\bar{j}$, transaction i obeys P3 with respect to transaction j,

then all possible interleavings of transactions in classes in $\underline{C}$ are serializable.

2.2.8.4 P2: A Faster Protocol for Read-Only Transactions

While P3 is sufficient for synchronizing all diagonal edges in a cycle, we can do somewhat better with those transactions that intersect the cycle only with their r-nodes. These read-only transactions contribute to nonserializability only because they may observe certain WRITE messages being processed in reverse timestamp order.* Protocol P2 is a weaker version of P3 that prevents this situation and thereby provides a less expensive alternative for synchronizing such transactions.

Suppose, for example, that the edges $\langle r^{\bar{I}},w^{\bar{J}}\rangle$ and $\langle r^{\bar{I}},w^{\bar{K}}\rangle$ appear in a conflict graph cycle. To synchronize a cycle, we want each set of transactions whose classes lie on the cycle to be serialized in timestamp order. If $\langle r^{\bar{I}},w^{\bar{J}}\rangle$ and $\langle r^{\bar{I}},w^{\bar{K}}\rangle$ are on this path, then transactions i, j, and k (say) in $\bar{I}$, $\bar{J}$, and $\bar{K}$ must be serialized in timestamp order. This two edge path will prevent a timestamp

---

* Strictly speaking, these transactions need not be read-only. It is just that their write operations, if they have any, do not participate in a conflict graph cycle.

ordered serialization _only if_ transaction i observes WRITE messages from j and k in reverse timestamp order. For example, suppose TS_j and TS_k are the timestamps of j and k and $TS_j < TS_k$. If R^i_{alpha} precedes and conflicts with W^j_{alpha} and R^i_{beta} follows and conflicts with W^k_{beta}, then from i's viewpoint and according to rules (1') and (2'), k must be serialized before i which must be serialized before j. If either R^i_{alpha} had followed W^j_{alpha} or R^i_{beta} had preceded W^k_{beta}, j and k could have been serialized in timestamp order. Protocol P2 is designed to make precisely this guarantee.

A transaction i obeys protocol P2 with respect to transactions j and k if for any alpha

1. if R^i_{alpha} precedes and conflicts with W^j_{alpha} and $TS_k > TS_j$, then R^i_{beta} precedes W^k_{beta} at every DM_{beta} where they both appear and conflict, and

2. if R^i_{alpha} follows and conflicts with W^j_{alpha} and $TS_j > TS_k$, then R^i_{beta} follows W^k_{beta} at every DM_{beta} where they both appear and conflict.

That is, if $TS_j < TS_k$ then transaction i observes a WRITE message from transaction k only if it has observed all WRITE messages from transaction j, and conversely if $TS_k < TS_j$. Protocol P2 prevents i from observing a WRITE message from the later transaction unless it has observed all WRITE messages from the earlier one.

Protocol P2 is strictly weaker than P3 in that if i obeys P3 with respect to j and k then it obeys P2 with respect to j and k. Yet we can use it correctly for synchronizing classes which only intersect cycles with their r-nodes. Stated precisely, if $[\langle w^{\bar{j}}, r^{\bar{i}}\rangle, \langle r^{\bar{i}}, w^{\bar{k}}\rangle]$ is a subpath of a cycle, then if for each i, j, and k in $\bar{i}$, $\bar{j}$, and $\bar{k}$ we have that i obeys P2 with respect to j and k, then we need not synchronize these two diagonal edges using P3.

2.2.9 A Summary of the Protocol Selection Rules

In Section 2.2.8, we described the three basic protocols for synchronizing transactions and the conflict graph topologies that require the use of the protocols. While the analysis that leads to the protocols is somewhat complex, the rules for selecting the protocols are not. It is these Protocol Selection Rules that completely govern the concurrency control mechanism of SDD-1. We present these rules here in order to summarize and encapsulate the results of Section 2.2.9 and to incorporate a few more details to make the statement of the rules precise.

First, let us restate each of the three protocols.

Protocol P1: Transaction i obeys protocol P1 with respect to transaction j if for each DM, alpha, if W^{i}_{alpha} is processed before (resp. after) and conflicts with R^{j}_{alpha}, then W^{i}_{beta} is processed before (resp. after) R^{j}_{beta} at every DM_{beta} where they both appear and conflict.

Protocol P2: Transaction i obeys protocol P2 with respect to transactions j and k if for any alpha:

1. if R^{i}_{alpha} is processed before and conflicts with W^{j}_{alpha} and k has a later timestamp than j, then R^{i}_{beta} is processed before W^{k}_{beta} at every DM_{beta} where they both appear and conflict, and

2. if R^{i}_{alpha} is processed after and conflicts with W^{j}_{alpha} and j has a later timestamp than k, then R^{i}_{beta} is processed after W^{k}_{beta} at every DM_{beta} where they both appear and conflict,

Protocol P3: Transaction i obeys protocol P3 with respect to transaction j if for each DM_{alpha} at which R^{i}_{alpha} and W^{j}_{alpha} both appear and conflict, R^{i}_{alpha} and W^{j}_{alpha} are processed in timestamp order.

Briefly, these protocols serve the following purposes:

P1: Prevents READ messages from one transaction that conflict with WRITE messages from another transaction from being processed in different relative orders at different DM's.

P2: Prevents a READ message from seeing WRITE messages from two other transactions in reverse timestamp order.

P3: Prevents race conditions.

The Protocol selection rules state which protocols should be invoked by which transactions. They are:

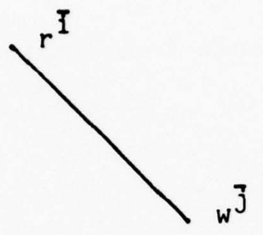
I. For all classes in $\bar{i}$ and $\bar{j}$ such that $<r^{\bar{i}}, w^{\bar{j}}>$ is in the conflict graph, for each pair of transactions i and j in $\bar{i}$ and $\bar{j}$ (respectively), i must obey protocol P1 with respect to j (see Figure 2.14a).

II. For each cycle in the conflict graph that contains a vertical edge, the following hold:

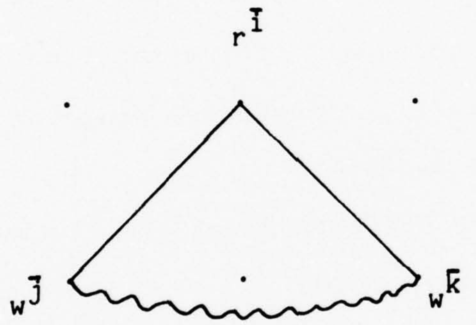
a. for all distinct classes $\bar{i}$, $\bar{j}$, $\bar{k}$, if edges $<r^{\bar{i}}, w^{\bar{j}}>$ and $<r^{\bar{i}}, w^{\bar{k}}>$ lie on the cycle, then for each set of transactions i, j, and k in $\bar{i}$, $\bar{j}$, and $\bar{k}$ (respectively), i must obey P2 with respect to j and k (see Figure 2.14a); and

b. for all distinct classes $\bar{i}$ and $\bar{j}$ such that $<r^{\bar{i}}, w^{\bar{j}}>$ and $<r^{\bar{i}}, w^{\bar{j}}>$ lie on the cycle, then for each pair of transactions i and j in $\bar{i}$ and $\bar{j}$ (respectively), i must obey P3 with respect to j (see Figure 2.14c).

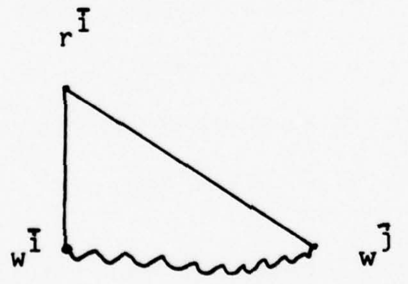
Protocol Selection Rules Figure 2.14



(a) For each i, j in $\bar{I}$, $\bar{J}$, i must obey P1 with respect to j.



(b) For each i, j, k in $\bar{I}$, $\bar{J}$, $\bar{K}$, i must obey P2 with respect to j and k.



(c) For each i, i in $\bar{I}$, $\bar{J}$, j must obey P3 with respect to j.

The protocol selection rules are easily transformed into an algorithm that analyzes the conflict graph and produces the protocols that each class must obey. However, the definitions of the protocols are not algorithmic. To make the protocols effective, we now show how TMs and DMs can enforce the relative orderings of READ and WRITE messages required by the protocols.

2.2.10 Implementing the Protocols

2.2.10.1 Implementing Protocol P1

Each protocol demands that certain relative orderings of READ and WRITE messages be obeyed. Protocol P1 demands that READ and WRITE messages of two transactions that correspond to the endpoints of a diagonal edge must be processed in the same relative order at all DMs where they are both processed. Suppose the diagonal edge is $\langle r^{\bar{i}}, w^{\bar{j}} \rangle$. Then P1 says that if there are two DMs, alpha and beta, such that R^i_{alpha} and W^j_{alpha} are processed and conflict at DM_{alpha} and R^i_{beta} and W^j_{beta} are processed and conflict at DM_{beta}, then R^i_{alpha} is processed before W^j_{alpha} iff R^i_{beta} is processed before W^j_{beta}.

Let us first examine a simple case. If transactions in class $\bar{i}$ only send READ messages to one DM at which conflicting WRITE messages from class $\bar{j}$ are processed, then P1 is trivially satisfied. Since only one DM ever processes conflicting messages, there is no chance for a different ordering of conflicting messages at different DMs. If class $\bar{i}$ sends READ messages to two or more DMs at which conflicting WRITE messages from class $\bar{j}$ are processed, then synchronization is needed. The synchronization information is carried entirely by the READ messages from $\bar{i}$ in the form of read conditions.

A read condition is attached to a READ message and specifies which WRITE messages from certain other classes must be processed before the READ message can be correctly processed. The read condition includes a timestamp, say TS, and one or more classes, say $\{\bar{j}_1, \ldots, \bar{j}_m\}$. The read condition tells the DM to hold the READ message until such time that all WRITE messages from classes $\{\bar{j}_1, \ldots, \bar{j}_m\}$ with timestamps prior to TS have been processed and that no WRITE messages from classes $\{\bar{j}_1, \ldots, \bar{j}_m\}$ with timestamps later than TS have been processed. Then the READ message can be processed.

To implement protocol P1 on $\bar{i}$ with respect to $\bar{j}$, a read condition $\langle TS, \{\bar{j}\}\rangle$ must be attached to each READ message sent on behalf of a transaction i in $\bar{i}$ to each DM at which conflicting WRITE messages from $\bar{j}$ are processed. This is sufficient to guarantee P1. For example, if R^{i}_{alpha} is processed after W^{j}_{alpha}, then transaction j must have a timestamp prior to TS. So, at any other site, say beta,

R^{i}_{beta} will be processed after W^{j}_{beta} since the same read condition applies there as well. Notice that the choice of the timestamp TS is immaterial to the correctness of the protocol. All that matters is that all read conditions associated with i have the same timestamp. As we will see in a moment, the choice of timestamp can affect the efficiency of the protocol.

To correctly process a READ message with read condition $<TS, \{\bar{j}\}>$ at a DM, the DM must wait until all WRITE messages from $\bar{j}$ with timestamps prior to TS have arrived and been processed. The class pipelining rule requires that WRITE messages from any given class be processed in timestamp order at every DM. So, as soon as the DM receives a WRITE message timestamped later than TS, it knows to hold it and process the READ message first. Of course, if a WRITE message from $\bar{j}$ with timestamp later than TS was processed before the read condition was received, then the READ condition cannot be satisfied without backing out the WRITE message. In SDD-1, no WRITE message is backed out for concurrency control reasons. So, in this case, the READ message would have to be rejected and the originating class must resubmit it with a later timestamp. Notice that all READ messages on behalf of transaction i have to be resubmitted, since their read conditions are now obsolete.

A problem with the mechanism described above is that class $\bar{j}$ may be idle because it has no transactions to process. The DM will therefore wait for a long time until a WRITE message timestamped later than TS arrives. One way to solve this problem is to have idle classes periodically send NULLWRITE messages.* A NULLWRITE message specifies the originating class and a timestamp and is interpreted as an empty WRITE message from that class with that timestamp. When a DM receives such a NULLWRITE message, it can be sure that it has received all WRITE messages from the indicated class through the given timestamp. If a DM chooses not to wait passively for a WRITE or NULLWRITE message from $\bar{j}$, it can request a NULLWRITE by sending a SENDNULL message to $\bar{j}$.

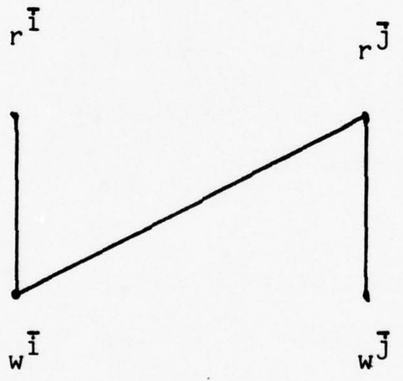
The choice of timestamps for read conditions and the rate at which NULLWRITEs are sent are important tuning parameters to avoid the frequent use of SENDNULLs. In addition, the choice of timestamp for read conditions will affect how long a READ message has to wait for conflicting WRITE messages to be processed. Essentially, the timestamp should be as small as possible without actually forcing the read condition to be rejected.

----

* The use of periodic NULLWRITE messages can be avoided by use of special protocols that are tailored for low

To illustrate the operation of protocol P1, let us consider a database that consists of two data items, x and y, where x is stored at DM_{alpha} and y is stored at DM_{beta}. Class $\bar{j}$ writes both x and y, and class $\bar{i}$ reads both x and y. For definiteness, suppose class $\bar{i}$ runs at $TM_{\bar{i}}$ and $\bar{j}$ runs at $TM_{\bar{j}}$. The conflict graph for this situation is shown in Figure 2.15. The edge $<r^{\bar{i}},w^{\bar{j}}>$ implies transactions in $\bar{i}$ must obey P1 with respect to

A Conflict Graph Illustrating P1 Figure 2.15



transactions in $\bar{j}$.

For $TM_{\bar{i}}$ to process a transaction, say i, it must send READ messages R^{i}_{alpha} to DM_{alpha} and R^{i}_{beta} to DM_{beta}. By P1, both messages must have a read condition $<TS,\{\bar{j}\}>$ attached. DM_{alpha} will not process R^{i}_{alpha} to read x until

frequency classes. However, their description is beyond the scope of this report.

it has received (but not processed) a WRITE message or a NULLWRITE from $TM_{\bar{j}}$ on behalf of $\bar{j}$ with timestamp later than TS. DM_{beta} will behave the same way. So, R^{i}_{alpha} will wait for (i.e., will be processed after) WRITE messages from the same set of transactions in $\bar{j}$ as R^{i}_{beta} will wait for. Hence, for each j in $\bar{j}$, rules (1') and (2') will require the same serialization order for i and j at both DM_{alpha} and DM_{beta}, and the result will be serializable. The nonserializable situation of R^{i}_{alpha} preceding W^{j}_{alpha} but R^{i}_{beta} following W^{j}_{beta} cannot occur.

2.2.10.2 Implementing P3

The same read condition mechanism that we described for implementing P1 is sufficient for implementing P3 as well. For transaction i to obey P3 with respect to all transactions j in $\bar{j}$ at DM_{alpha}, R^{i}_{alpha} must be processed after all W^{j}_{alpha} with earlier timestamps and before all W^{j}_{alpha} with later timestamps. If the timestamp of i is TS_i, then attaching the read condition $\langle TS_i, \{\bar{j}\} \rangle$ to R^{i}_{alpha} will force DM_{alpha} to process R^{i}_{alpha} according to P3; DM_{alpha} will wait for exactly those W^{j}_{alpha} with $TS_j < TS_i$.

From this implementation, we see immediately that protocol P3 is strictly stronger than protocol P1. If i obeys P3 with respect to j at DM_{alpha}, then i obeys P1 with respect to j at DM_{alpha}. The difference between P1 and P3 is that P1 allows any timestamp to appear in the read condition while P3 requires that timestamp to be TS_i.

Our earlier remarks about NULLWRITEs and SENDNULLs apply here as well. We noted under P1 that choosing a timestamp for the read condition was important to avoid lengthy delays. Since the read condition timestamp is the transaction's timestamp in P3, we must be careful to run the P3 transaction as early as possible -- early enough so that READ messages need not wait for many WRITE messages but not so early as to require its being rejected.

2.2.10.3 Implementing Protocol P2

As with the other protocols, P2 is implemented using read conditions. If i must obey P2 with respect to transactions in classes $\overline{j}$ and $\overline{k}$, then it must attach a read condition $<TS, \{\overline{j},\overline{k}\}>$ to each of its READ messages that are sent to a DM that processes conflicting WRITE messages from $\overline{j}$ or $\overline{k}$. As in P1, any timestamp for the read condition will do. Since some DMs will only process conflicting WRITE messages for either $\overline{j}$ or $\overline{k}$ (but not both) these DMs will only use one of the two classes in the second read condition parameter. If i conflicts with WRITE messages from $\bar{j}$ and $\bar{k}$ at *only one* DM, an interesting optimization is possible. Rather than specifying the timestamp TS in the read condition, the DM can select the timestamp itself. As long as there is *some* time, TS, such that all earlier WRITE messages and no later WRITE messages from $\bar{j}$ and $\bar{k}$ have been processed, P2 will be obeyed. However, if two or more DMs are involved, the timestamp must be fixed in advance, because all DMs must use the *same* timestamp; they cannot choose timestamps independently.

2.2.11 P4: A Cycle-breaking Protocol

Although P1, P2, and P3 are sufficient to guarantee serializability, from an efficiency standpoint these protocols have a very serious problem. The problem is that a single class can cause many cycles and thereby force many classes to use P2 and P3, *even though very few transactions are ever run in that class*.

While we expect that the vast majority of transactions that we wish to execute are predictable and belong to predefined classes, we still want to be able to execute an unexpected transaction that does not fit into any of our class definitions. One way to accomplish this is to define a very "large" class, call it $\bar{I}_{total}$, that has a read-set and write-set that includes the entire logical database. Every conceivable transaction can fit into $\bar{I}_{total}$, so this apparently solves the problem. But the cost is enormous, for $\bar{I}_{total}$ induces a two-class cycle with <u>every other class</u> in the system. So, every class has to run P3 against $\bar{I}_{total}$, and $\bar{I}_{total}$ has to run P3 against every other class. Since P3 is the most expensive protocol, this is an unfortunate state of affairs. It is especially unfortunate because transactions will rarely need to execute in $\bar{I}_{total}$, since most transactions fit into other less expensive classes. So, $\bar{I}_{total}$ introduces considerable synchronization overhead for synchronizing against a class that will rarely run a transaction.

In general, any class in which transactions are only infrequently run, but which creates many cycles in the conflict graph, exhibits this phenomenon. Although the problem of proliferation of cycles is especially acute in $\bar{I}_{total}$, other classes with smaller read-sets and write-sets may manifest the same problem.

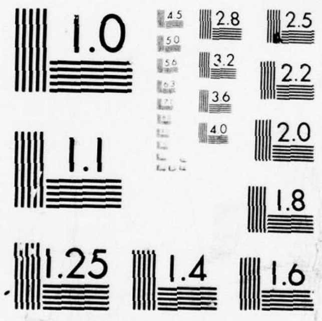
To alleviate these problems we introduce a new protocol called P4, the purpose of which is to "break" cycles in the conflict graph. That is, if a class runs P4, then

AD-A068 161 COMPUTER CORP OF AMERICA CAMBRIDGE MASS F/G 9/2
A DISTRIBUTED DATABASE MANAGEMENT SYSTEM FOR COMMAND AND CONTRO--ETC(U)
JAN 79 N00039-77-C-0074

UNCLASSIFIED CCA-79-12 NL

2 OF 2
AD
A068161

END
DATE
FILMED
6--79
DDC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

other classes that are in a cycle with the P4 class can behave as if the cycle did not exist.

One way to implement P4 is to shut off the system when a P4 transaction is introduced. No new transactions are processed and the system works until all outstanding WRITE messages from transactions already in progress have been processed. When the system has finally quiesced, we can safely run the P4 transaction serially. After all of the P4 transaction's WRITE messages are processed, we can safely permit the system to process transactions again. Since the execution before and after the P4 transaction ran was serializable (by the serializability theorem) and since the P4 transaction ran serially, the entire execution is serializable.

Of course, this implementation is likely to be unacceptable due to the severe performance degradation that results from shutting off the system, even temporarily. To improve the protocol, we first observe that a P4 transaction need only synchronize against classes that lie on a cycle that includes the P4 class, since only classes on cycles can cause nonserializability. Second, we note that even these classes need not quiesce completely before running a P4 transactions. Only <u>conflicting</u> WRITE messages must be completed before the P4 transaction executes and allows the other classes to resume processing. WRITE messages that do not conflict with READs in the same cycle cannot affect the ordering of transactions in the serialization according to rules (1')-(3'), and therefore they do not require synchronization under P4.

The implementation of P4 differs structurally from the other protocols in two ways. First, P4 requires some direct communication between TMs. By this communication, the P4 class requests that certain other TMs perform synchronization to avoid conflicting with the P4 transaction. Second, P4 requires an augmented form of read condition. Recall that a standard read condition is a pair of the form <timestamp, {classes}>. For P4, the timestamp may be interpreted as a "minimum time", i.e., <mintime=timestamp, {classes}>. This condition is satisfied if all WRITE messages from {classes} timestamped less than "timestamp" have been received. It does <u>not</u> require that no messages from {classes} timestamped greater than "timestamp" be received (as in standard read conditions).

To implement P4, we use three additional types of messages that are sent from TMs to TMs (<u>not</u> from TMs to DMs). A P4-ALERT message is sent from a P4 class to some other class. A P4-ALERT message includes the name of the P4 class and the timestamp of the P4 transaction as its parameters. A class responds to a P4-ALERT with either a P4-ACCEPT or a P4-REJECT.

To run a transaction i_{P4} in the P4 class $\bar{i}_{P4}$, one performs the following steps:

1. Choose a timestamp for i_{P4}, say TS_{P4}.

2. Send a message P4-ALERT (TS_{P4}) to every class that lies on the cycle with $\bar{i}_{P4}$ in the conflict graph.

3. Wait for the P4-ACCEPTs to be received from all classes to which a P4-ALERT was sent. If a P4-REJECT is received, then restart the protocol from step 1.

4. Construct the READ message for i_{P4}. For each DM_{alpha} and class $\bar{j}$ such that $\langle r^{\bar{i}}P4, w^{\bar{j}}\rangle$ lies on a cycle and $\bar{j}$ sends WRITE messages to DM_{alpha} that can conflict with $R^{i_{P4}}_{alpha}$, attach the read condition $\langle TS_{P4}, \{\bar{j}\}\rangle$ to $R^{i_{P4}}_{alpha}$.

When a TM receives a P4-ALERT (TS_{P4}) for a particular class, $\bar{j}$, it performs the following steps:

1. If $\bar{j}$ has run or begun running a transaction with a timestamp greater than TS_{P4}, then respond to $\bar{i}_{P4}$ by sending P4-REJECT. Otherwise, send P4-ACCEPT and do not run another transaction in $\bar{j}$ timestamped earlier than TS_{P4}.

2. In processing the next transaction run in $\bar{j}$, say j, for each DM_{alpha} to which j sends a READ message and for each class $\bar{k}$ such that $\langle r^{\bar{j}}, w^{\bar{k}} \rangle$ lies on a cycle with $\bar{i}_{P4}$ and $\bar{k}$ sends WRITE messages to DM_{alpha}, attach the read condition $\langle mintime = TS_{P4}, \{\bar{k}\} \rangle$ to R^{j}_{alpha}. These conditions are in addition to those normally carried by R^{j}_{alpha}. (Note: Only do this step for the <u>first</u> transaction in $\bar{j}$ with timestamp later than TS_{P4}.)

2.2.12 The Concurrency Monitor

The implementation of the run-time concurrency control mechanism primarily lies in a software module at the DMs called the Concurrency Monitor. The Concurrency Monitor at a DM accepts READ, WRITE, and NULLWRITE messages from TMs and schedules their execution at the DM. In essence, it is responsible for determining the ordering of events for local DM logs. In this section we will describe the operation of the Concurrency Monitor. As we will see, the mechanism is quite simple.

The Concurrency Monitor accepts and schedules messages of three types:

WRITE (TS, CLASS, UPDATES)

> TS is the timestamp of the transaction issuing the WRITE, and CLASS is its transaction class. UPDATES is a list of data item identifiers and values. When a WRITE is processed, the indicated data items are updated to the specified values according to the WRITE Message Rule (see Section 2.2.4.)

NULLWRITE (TS, CLASS)

> This message indicates that all future messages in CLASS will have timestamp greater than TS. Processing the NULLWRITE simply involves taking note of this fact in the internal tables of the Concurrency Monitor.

READ (TS, CLASS, READSET, CONDITIONS)

TS and CLASS are the timestamp and transaction class of the transaction issuing the READ message. CONDITIONS is a list of read conditions associated with the READ message. Processing a READ involves reading the current values for data specified by READSET into a local transaction "workspace".

The read conditions have the following format:

<TYPE, CLASSES, TS>

CLASSES is a list of transaction classes. TS is either a timestamp or is blank, depending on TYPE. If TYPE is "normal", then the read condition is satisfied when all WRITE messages from the listed classes with timestamps less than TS have been processed, but no WRITE messages from those classes with greater timestamps have been processed. "Normal" read conditions are used in all four protocols. If TYPE is "DMchoice", then the TS specification is blank; the read condition is satisfied when the condition for "normal" read conditions can be satisfied for some selected value for TS. "DMchoice" read conditions are used in protocol P2. If type is "mintime", then the read condition is satisfied when all WRITE messages from the listed classes with timestamps less than TS have been processed. "Mintime" read conditions are used in the P4 protocol. The TS specification in a read condition is always less than the transaction TS specified in the READ message itself.

The DM returns an ACCEPT-READ message when all the read conditions on a READ message have been satisfied and the READ has been processed. If the read conditions cannot be satisfied, even by waiting for new WRITE messages to be processed, then a REJECT-READ message is returned to the originator of the READ.

The function of the Concurrency Monitor is to schedule the processing of READ and WRITE messages under the constraints imposed by read conditions. READ messages can be processed as soon as they are satisfied. While WRITE messages should be processed without unnecessary delay, a WRITE message will be delayed if its immediate processing would cause the rejection of a pending READ message. When a READ message is received, it is checked to see if it is immediately rejectable. If it is not, then the READ will eventually be satisfied, because the Concurrency Monitor will not process any WRITE messages that will cause it to be rejected.

The Concurrency Table shown in Figure 2.15.1, contains the information needed by the Concurrency Monitor to resolve the status of read conditions. For each class, it holds a timestamp associated with the most recently processed WRITE or NULLWRITE message and a pointer to a queue of pending messages from that class to be processed. Within each queue, READ and WRITE messages appear in increasing timestamp order. This follows from the pipelining rules, and the fact that messages are guaranteed by the network to be received in the same order that they were sent. The Concurrency Monitor schedules the messages on each queue in the order that they appear. The message at the head of the queue is said to be _immediately pending_.

The Concurrency Table Figure 2.16

Class	timestamp of most recently processed WRITE	timestamp of most recently processed NULLWRITE	pointer to pending message queue
i	425179	425221	----------->
.	.	.	.
.	.	.	.
.	.	.	.

The Concurrency Monitor chooses the next message to be processed based on the following criteria:

1. Process any pending NULLWRITE.

2. If there are none, process any immediately pending WRITE, as long as this does not cause any pending READ to be rejected.

3. If there are no such WRITEs, process any immediately pending READs whose read conditions are satisfied.

It is important that the Concurrency Monitor not indefinitely postpone the processing of any immediately pending message either due to timing anomalies or deadlock. One way to guarantee this would be to schedule immediately pending messages according to the following priority rule. The priority of an immediately pending NULLWRITE or WRITE message is the TS parameter in the message; for a READ message, it is the lowest timestamp in an unsatisfied read condition in the READ. The Concurrency Monitor schedules smallest-priority-first.

We need to show that the smallest priority message can be processed within finite time. Let M be the message with lowest priority. If M is a NULLWRITE, it can be processed immediately. If M is a WRITE, then it will be held up only if there is an immediately pending READ with a read condition that has a timestamp smaller than M's. But then the READ would have a smaller priority than M, contradicting the choice of M. So, the WRITE can be immediately processed. Suppose that M is a READ. If it can be immediately processed then we are done. So, assume not and that $\langle TS_R, \{\bar{I},\ldots\}\rangle$ is its unsatisfied read condition with lowest timestamp. Let M' be the immediately pending message on $\bar{I}$'s queue. If the queue is empty, a WRITE or NULLWRITE message with timestamp greater than TS_R will eventually appear on $\bar{I}$'s queue, since there are only a finite number of timestamps smaller than TS_R. If M' is a WRITE, then M' must have a timestamp greater than TS_R (by choice of M) and the READ condition is already satisfied, a contradiction. Similarly, M' cannot be a NULLWRITE. If M' is a READ, it must have an unsatisfied read condition $<TS'_R, \{...\}>$ with $TS'_R > TS_R$ (by choice of M). By pipelining rule C3, any WRITE message following M' has timestamp greater than M', and, as mentioned earlier, the timestamp of a transaction must be greater than the timestamp of its read condition. So, by transitivity, every WRITE message from $\bar{I}$ will have a timestamp greater than TS_R and again $<TS_R, \{\bar{I},...\}>$ is satisfied, a contradiction. Hence, the READ can be processed immediately. Since there are only a finite number of timestamps less than any priority, this also argues for proper termination, since every message will eventually be the one with the lowest priority.

It may not be wise to strictly follow this priority rule, since a lowest priority READ may wait for a while for all the necessary WRITEs to arrive. This would unnecessarily create a large backlog of other unprocessed messages. However, the above argument demonstrates feasibility; of course, any more efficient variation which never indefinitely postpones is also acceptable.

2.2.13 Advantages of the SDD-1 Concurrency Control Mechanism

The SDD-1 approach to concurrency control is in many ways quite different from other proposed mechanisms. We see many strengths in the approach. Unfortunately, there are few analytic methods for verifying these strengths, say by comparing the relative performance of our mechanism to other database concurrency controls. Furthermore, most of the proposed mechanisms are not yet implemented, so empirical comparisons are not possible either. Hence, the analysis of our mechanism must necessarily be more intuitive than mathematical. The specific criteria on which we base performance comparisons include: the amount of communication required to synchronize transactions; the average delay incurred by a transaction due to concurrency control; the amount of concurrency among transactions allowed by the concurrency control; and the overhead involved in making the mechanism resilient to communications and node failures.

At the architectural level, the SDD-1 concurrency control mechanism has two important properties. First, the architecture makes a strong separation between concurrency control issues and those of query processing and reliability. From a project management standpoint, this separation allowed us to attack the concurrency control problem independently from and in parallel with query processing and reliability problems. From a software engineering standpoint, this division of labor led naturally to a division of function in software components. The concurrency control mechanisms are isolated in a small number of modules, making them easily modifiable and tunable.

Second, the architecture fully distributes the concurrency control. While each transaction is controlled from a single site, different sites are concurrently supervising the synchronization of many different transactions. No one site is in charge of <u>any</u> system-wide activity. The main advantage of this full distribution is enhanced reliability. A site failure only affects those transactions executing and/or using data at that site.

However, it is in the specific synchronization mechanisms that the most important advantages lie: conflict graph analysis and the timestamp-based protocols. We believe the technique of conflict graph analysis to be our most important contribution. By preanalyzing transaction conflicts, the number of transactions that need to be synchronized is drastically reduced. This has a beneficial effect on all aspects of concurrency control performance. It allows more concurrency among transactions; and for those transactions that require little or no synchronization it cuts delay, communications overhead, and costs associated with resiliency mechanisms. As shown in [BERNSTEIN and SHIPMAN b], the technique is quite general and can be used with a variety of synchronization protocols, including conventional locking. In principle, every proposed concurrency control mechanism could be improved by adding conflict graph analysis as a preprocessing step to eliminate run-time synchronization for some transactions.

The timestamp-based protocols, {P1,P2,P3,P4}, also offer important advantages over other proposed concurrency controls. First, the use of timestamps to resolve races among transactions eliminates the possibility of deadlock. Deadlock detection must be incorporated in any locking system and induces communications costs that the SDD-1 mechanism avoids. Second, the protocols synchronize transactions only against <u>named</u> transaction classes. Even if two transaction classes must be synchronized relative to certain data, other classes can concurrently access that data; in fact, other classes can independently be synchronized against that very same data without affecting the first two classes at all. This is in contrast to locking protocols, which set blanket locks that apply to all transactions that access the shared data. Third, SDD-1 offers a range of synchronization protocols. Protocol P2 is a fast synchronization protocol for read-only transactions that can afford to read an old, but consistent copy of the database. While with a locking strategy read-only transactions could choose not to lock the data they read, that unlocked data may be inconsistent. Protocol P4 allows infrequently executed transactions to take a larger share of the synchronization burden. By running such transactions under P4, other frequently executed transactions can run P1 with less delay and more concurrency than they would obtain if they ran P2 or P3 as otherwise required. The P4 capability is currently unique to the SDD-1 mechanism.

2.3 Reliability Mechanisms in SDD-1

We deal in this section with the problem of assuring the continued and correct operation of SDD-1 in the face of failures of sites or communications facilities. These mechanisms divide naturally into three domains: those concerned with the reliable operation of the query processing mechanism, those concerned with the reliable operation of the concurrency control mechanism, and those which seem to derive from the distributed environment and are not peculiar to SDD-1. These latter mechanisms are structured into a layer of software called the "reliable network".

2.3.1 Mechanisms in the Reliable Network

2.3.1.1 The Network Clock

The Reliable Network contains the clock used by SDD-1 to assign timestamps. This clock is also used internally by the Relnet for message timestamping.

The network clock is a logical clock. The only guarantee is that the timestamps it generates are unique and that they are assigned in monotonically increasing order. Thus, the clock is advanced after each timestamp assignment. The clock may also be "bumped" to a particular time, skipping over the intermediate timestamps, but it may never be retarded. To insure that timestamps are globally unique across the network, the local site number is appended as the timestamp's lowest-order bits.

For purposes of efficient operation of SDD-1's concurrency protocols, however, it is desirable for the clocks at all sites to be synchronized as closely as possible. To this end, a real-time clock is also maintained. The real-time clocks are assumed to be synchronized. The logical clock used for timestamping is always kept ahead of the real-time clock. This is accomplished by bumping the logical clock to the real-time clock value at each real-time clock tick. Thus, the logical clocks at each site will differ from real-time by the number of timestamps assigned since the last real-time clock tick. By making the logical clock increment a small enough fraction of a real-time clock tick, the logical clock can be kept arbitrarily close to the real-time clock. Below, any references to "the clock" will be to the logical timestamping clock, not the real-time clock.

We utilize the technique proposed by [LAMPORT] to coordinate the logical clocks at each site. In this scheme, each message, as it is being sent, is assigned a timestamp. This message timestamp is read by the receiving site and its local clock is bumped to have at least that value. By use of this technique, we can guarantee that if one message is logically dependent upon another message in the system, it will have a greater timestamp.

2.3.1.2 Message Acknowledgement

Upon receipt of a message, the receiving Relnet software will place the message on secure storage (e.g. disk) and then send an <Ack> back to the sending site. The sender Relnet can then discard its local copy of the message and inform its caller that the message has been delivered. (Note: This acknowledgement is distinct from the Arpanet level acknowledgement used to prevent lost messages.)

2.3.1.3 Status Monitoring

A site is considered "down" when it fails to acknowledge a message within a specified timeout period. Sites may also be marked down explicitly by the TM or DM software, if its behavior seems to be unusual for any reason. When a site is marked down, it is sent a <You're Down> message. Upon receipt of such a message (if the site is in fact up) it must stop all transactions in progress and simulate a failure followed by recovery. This harsh requirement insures that sites are uniformly considered down by all sites in the network, rather than down by some and simply sluggish by others.

Recovering sites send an <I'm up> message to all other sites. This causes the site to be marked "up".

A process using the Relnet may request that a site be failure watched. If the site fails, the process will be informed of this within a specified time after the failure. This must occur even if there is no ordinary message traffic with the site. Failure watching is accomplished with the use of <Probe> messages. After a given probe interval has elapsed since the last communication with the site, a <Probe> message is sent. The <Probe> is considered a no-op, but it must be acknowledged. Lack of acknowledgement for a <Probe> message indicates the site has failed.

A process with an outstanding failure watch on a site will also be informed of the site's failure if an <I'm up> message is received. Receipt of the <I'm up> signals that the site had failed and recovered before a probe message was able to detect the failure.

A recovery watch may also be requested against a site. In this case, the requesting process will be informed upon receipt of an <I'm Up> message from the site.

2.3.1.4 Spoolers

2.3.1.4.1 Spooler Overview

Each site has a designated list of spooler sites. When a site is down, any messages destined for it are sent to the spooler sites instead. At the spooler sites, a process, called the <u>spooler</u>, will receive the message and buffer it. The spooler behaves as a remote input buffer for the failed site. When the failed site recovers, it retrieves its messages from the spoolers.

By designating more than one spooler per site, the DBA can increase the likelihood that, even in cases where the spoolers themselves fail, it will be possible for the recovering recipient to retrieve all of its messages.

Because it is possible for spoolers to fail and recover while they are spooling and because it is possible for spoolers or the recovering recipient to fail during the despooling process, the algorithm for despooling is somewhat intricate.

2.3.1.4.2 The Last Message Vector

To support the spooling mechanism, each site maintains a Last Message Vector. This is a vector with one entry per site that records the timestamp of the last message received from that site. Since Arpanet level mechanisms prevent messages from being received out of order, we can assume that any message that is received from a site with timestamp less than or equal to that recorded in the Last Message Vector is a duplicate of an earlier message. Such messages are acknowledged but are subsequently ignored.

2.3.1.4.3 Basic Spooling Algorithm

We first describe the basic algorithm which operates in the absence of such untimely failures. Upon recovery, the recipient site issues an <I'm up> message to all other sites. Those sites which had been spooling messages for the recovering site stop spooling. The transmitting sites will now send any new messages directly to the recipient. If a transmitting site had been in the middle of spooling a message when the <I'm up> was received, it sends that message directly to the recipient. This message may or may not also be found in some of the spoolers, but the system has been designed to eliminate duplicate messages in any case. The recovering site then chooses a spooler and sends it a <Despool> message. The spooler will forward the messages which it has been buffering. The recovering recipient will place these messages in its input queue and may begin processing them. During the despooling process, messages which are directly received from a sender are kept in an intermediate despooling buffer. When the spooler has sent all its buffered messages, it sends the recipient an <Empty> message. Upon receipt of the <Empty> from the spooler, the recipient will begin removing messages from the despooling buffer. When the despooling buffer has emptied, normal message reception resumes with messages being placed directly into the input buffer at the recipient site.

Rather than make use of the despooling buffer, we could have chosen to have the senders hold their messages during despooling, sending them to the recipient only after the spoolers had been emptied. This makes for a conceptually simpler approach, but has the side-effect of keeping messages at the sender sites unsent, and hence unacknowledged, for perhaps quite long periods of time. So long as its update messages are unacknowledged, a transaction cannot complete. And other transactions waiting on the completion of that transactions are also held up. Therefore the approach of holding up messages at the sender sites was rejected.

Another approach would be for the sender sites to continue spooling messages while despooling was in progress. Because the recipient will presumably be able to despool faster than the senders can spool (or else the system would clog during normal operation), the spooler will eventually empty. This alternative, however, is somewhat more intricate as well as having the disadvantage that the messages which are spooled during despooling will be sent through the network twice.

It is worth noting at this point that the various spoolers for a recipient will not necessarily have identical message streams. Because messages are being received from different sources, it is possible for messages have different orders in different spoolers. For example, if A and B are sending a message concurrently to two spoolers S1 and S2, it is possible for A's message to arrive first at S1 and B's message to arrive first at S2. It should not matter, however, which spooler is chosen for despooling, because either contains an ordering which <u>could have occurred</u> at the recipient had it been up.

2.3.1.4.4 Spoolers Which Have Crashed

One problem not addressed by the basic scheme is the situation where some spoolers are down at the time despooling begins. In this case those spoolers are simply ignored and are not sent <Despool> messages. If all spoolers are down, then it will not be possible for the recovering site to obtain its messages. Hopefully, spoolers will have been chosen so as to make this a highly unlikely occurrence.

Another problem occurs in the case of spoolers going down while they are despooling. In this situation a new spooler is chosen to despool from and is sent a <Despool> message, extended to include the Last Message Vector current at the time when the <Despool> is sent. The spooler will examine each message before sending it and will discard any message which has already been received by the recovering recipient, thus eliminating the need to retransmit messages which had previously been forwarded by another spooler.

2.3.1.4.5 Spoolers Which Have Crashed and Recovered

A third situation not addressed by the basic algorithm is one in which spoolers have failed and recovered during the period in which the destination site was down. In this situation, the spooler will have gaps during which it was not able to buffer messages for the destination. Our solution to this problem is to have spoolers which have failed insert a special <Gap> message in their message streams when they recover. This <Gap> message will indicate the position in the message stream during which the spooler was down.

While despooling, if the spooler encounters a <Gap> message in its message stream, it will send a <Gap> message to the recovering destination site. The <Gap> message will remain in the message stream, however. After sending the <Gap> message the spooler stops despooling. The destination site then chooses a new spooler to despool from. It sends the newly chosen spooler a <Despool> message containing the current Last Message Vector. As previously discussed, messages are discarded based on the Last Message Vector. A <Gap> message may be discarded if all previous messages are discardable and if the following message is discardable. Despooling begins with the first message which cannot be discarded. Note that this may be a <Gap> message.

The recovering site's algorithm is now as follows. A spooler is chosen and <Despool> is sent, along with the current Last Message Vector. When and if a <Gap> message is encountered, then a different spooler is chosen. <Despool> and the current Last Message Vector are sent to it. Subsequent <Gap> messages cause a new spooler to be chosen for despooling. If all of the spoolers send <Gap> messages when given the same Last Message Vector, then there must have been a period of time when all of the spoolers were down. The messages sent during that period are lost. Again, hopefully spoolers will be chosen so that this is an extremely rare occurrence. More likely, however, is that an <Empty> message will eventually be received from some spooler. At that time the despooling buffer is emptied and normal operation proceeds as previously described.

2.3.1.4.6 Aborted Recovery Attempts

A fourth problem not addressed by the basic despooling strategy is what happens when the recovering recipient fails while it is in the midst of the despooling operation. The difficulty here is that the sending site will have been sending messages directly to the recipient's despooling buffer and will, upon failure of the recipient, begin spooling its messages again. In order to reconstruct the correct message sequence, messages from the despooling buffer will have to be interspersed into the message stream from the spoolers.

This problem is solved as follows. Upon recovering, a timestamp is obtained and designated the <u>Recovery Timestamp</u>. This Recovery Timestamp is unique and serves to identify recovery attempts made by the recovering recipient. After coming up, the recovering recipient puts a <u>Recovery Marker</u> at the end of its despooling buffer. The Recovery Marker is flagged with the current Recovery Timestamp. It then sends <I'm up> messages to all the sending sites. After these are acknowledged, it sends a <Recovering> message to all its spoolers. The <Recovering> message contains the Recovery Timestamp for this recovery. Upon receipt of the <Recovering> message, each spooler will place an <Empty> message, flagged with the Recovery Timestamp, at the end of its message queue. Thus, <Empty> messages will be sent automatically because they appear explicitly in the spooler's message stream.

Now the recovering recipient performs the despooling procedure as previously described. When an <Empty> message is received, however, it checks whether the Recovery Timestamp with which the <Empty> message is flagged is the same as the current Recovery Timestamp. If it is, then the despooling buffer is drained of its messages and message communication proceeds as normal. If it is not the same, then the <Empty> message had been placed during an earlier, aborted, recovery attempt. In this case, the despooling buffer is emptied only up to the first Recovery Marker with Recovery Timestamp greater than the one associated with the <Empty> message which had been received. This will retrieve exactly those messages placed in the despooling buffer during the aborted recovery attempt. At this point, the despooling procedure simply continues despooling from the selected spooler until eventually an <Empty> message with the appropriate Recovery Timestamp is received.

Note that <Empty> messages in the spoolers and recovery markers in the despooling buffer are linked by their associated Recovery Timestamps. Thus, despooling following aborted recovery attempts will proceed correctly, even if the recovering recipient had crashed in the middle of sending <Recovering> messages to its spoolers.

This final version of the algorithm should cover all the possible failure/recovery situations which can cause despooling problems.

2.3.1.5 Transaction Control

A transaction is a global operation which may be treated as an atomic, indivisible operation by users of the system. It is the system's responsibility to insure that partial effects of a transaction are not observed or recorded in the database.

The query processing mechanism insures that updates are made to all copies of all data items updated by a transaction. The only way for the effect of a transaction to be only partially recorded would be for the transaction to fail after some updates had been distributed but before all of them had. It is the function of the transaction control component of the Relnet to deal with this problem.

The concurrency control protocols guarantee that it will not be possible for a transaction to <u>observe</u> the partial effects of other transactions. This is accomplished by having transactions wait until all updates from a transaction have taken place. If a part of a transaction has been lost, then the concurrency control algorithms will hang waiting for it to arrive.

The approach used by the Relnet to solve this problem is based on the use of <Commit> messages. The updates made by a transaction will not actually be recorded in the database until a <Commit> message is received, and the <Commit> messages are not sent until all updates have been distributed. Our approach is an extension of a traditional technique known as two-phase commit (see [GRAY]).

2.3.1.5.1 Basic Commit Algorithm

The global coordination of the various processes needed to execute a transaction is performed by the transaction's controlling TM. The controlling TM extracts the information needed to process the request and collects it at one site, called the final DM site (see [WONG et. al.]). The request is then run at the final DM site. If the request is an updating request, then a log of database updates is kept during the request execution. This log is transformed into a series of <Write> messages, which are sent to all sites which hold copies of the updated data items. The <Write> messages may be interpreted as commands to the foreign sites to initiate an update process on behalf of the transaction to perform the local update.

All updates, those at the final DM site as well as those at sites which are responding to <Write> messages, are performed in such a way that their effects are not seen in the database until they are committed. Updates may also be aborted, in which case the update is disposed of and no change to the database occurs. This is accomplished through the use of a differential file technique (see [SEVERENCE and LOHMAN], [EASTLAKE]).

After the final DM site has completed sending all of the <Write> messages for a transaction, it sends a completion response to the TM controlling the transaction. The TM will then send a <Commit> message to all of the sites where updating for the transaction is to be performed. Upon receipt of the <Commit>, the DMs will commit all of the updates for that transaction.

The TM will have a failure watch outstanding on the final DM site while it is processing the request. If the final DM site crashes, the TM will discover this and will send <Abort> messages to all of the sites which might have received an update for the transaction. The <Abort> will be ignored at a site if the update had not yet been requested there. Thus, if some, but not all, of the <Write> messages from a transaction are sent at the time the transaction fails, then none of the updates will be committed.

The controlling TM site itself might fail before all of the <Write> or <Commit/Abort> messages are sent out. Because of this, a DM which expects a <Commit/Abort> message (either because it is a final DM site or because it has received an <Write> message), will Failure Watch the TM executing the transaction. If that TM goes down before the <Commit/Abort> is received, then the DM will initiate a procedure to determine for itself whether or not the update should be committed.

This commit resolution proceeds as follows. After noticing that the controlling TM has failed, the DM will query all of the other sites which should have received the <Commit/Abort> message from the TM to determine whether or not any of them have in fact received the <Commit/Abort> message. If any have, then this indicates that a <Commit/Abort> message would also have been sent to the querying DM, had the controlling TM stayed up long enough, and hence the update should be committed. On the other hand, if none of the other DMs had received a <Commit/Abort> message, then the update is aborted. Note that so long as none of the DMs involved fail during the commit resolution procedure, then all of them will arrive at the same decision as to whether to commit or abort their updates.

To support the commit resolution procedure, the final DM site includes with the <Write> messages it sends, the identity of the controlling TM and the list of DM sites which should expect to receive <Commit/Abort> messages.

2.3.1.5.2 Delayed Commits

It is possible that, immediately after the controlling TM sends the first <Commit/Abort> message to a DM, both the TM and that DM crash. In this case, it will not be possible for other DM sites which are expecting <Commit/Abort> messages to determine, during their commit resolution procedure, that a <Commit/Abort> had actually been sent.

To deal with this issue, a commit delay may be associated with a <Commit> message. The commit delay has the effect that the update in question will not be committed until a specified real-time delay interval has passed since the receipt of the <Commit> message. If the DM site receiving the <Commit> crashes before the interval has passed, then upon recovery, the <Commit> is discarded and the DM must perform commit resolution (as if it had never received the <Commit> message).

If, during commit resolution, a DM is queried that has received a <Commit> but for which the commit delay interval has not yet elapsed, then the response to the query will include the remaining delay time interval. The querying DM will then behave as if it had received a <Commit> with that time interval.

The purpose of the commit delay mechanism is to prevent the dangerous effect described earlier where the only site that had received the <Commit> message crashed shortly thereafter so that proper commit resolution by other sites was not possible. The problem was that the one site would commit the update but the others, because they could not learn of this, would decide to abort the update. With the appropriate commit delay, the problem would not occur since the one site which received the <Commit> message would not have actually committed the update at the time of the crash. Upon its recovery it would query the other DMs involved and learn that it should in fact abort the update. In order to insure proper operation, the commit delay should be set long enough to enable at least one of the other DMs to query it before it crashes.

It is not necessary to send commit delays on every <Commit> message. A reasonable strategy would be for only the first few <Commit> messages to have commit delays, since after a number of <Commit> messages have been sent it becomes very unlikely for commit resolution to fail. The number of <Commit> messages which have commit delays, as well as the length of the delays, are an adjustable system parameter. Fewer or shorter commit delays will increase the likelihood that the commit resolution procedure will operate incorrectly.

The commit delays have the undesirable effect of delaying the actual updating operations of a transaction and consequently delaying the initiation of any transactions which must wait for that transaction to complete. The result being a decline in system throughput. To alleviate this, the controlling TM, after sending a number of <Commit> messages, may re-send <Commit> messages without commit delays to those sites which previously had received commit delays. This would allow the DM to proceed immediately to commit the update.

2.3.1.5.3 Auxiliary Commit Sites

An alternative approach to the throughput problem introduced by commit delays is the use of <u>auxiliary commit sites</u>. These are sites which are placed on the commit list but which do not have pending updates for the transaction being committed. They are sent <Commit> messages, however, and are queried during a commit resolution. No commit delay need be sent with the <Commit> messages to updating sites, providing <Commit> messages with commit delays are first sent to the auxiliary commit sites. Since no update is pending at the auxiliary commit sites, the commit delays there do not slow system throughput.

2.3.1.5.4 Interaction with Spooling Mechanism

<Commit/Abort> messages are sent to spoolers in the same way other messages are. Upon despooling the receiving site will treat the <Commit/Abort> as if it had received it directly from the sending TM.

To simplify the spooling mechanism, however, spoolers will not be required to respond to commit resolution queries. So far as the spooling mechanism is concerned, <Commit/Abort> messages are no different that any other type of message that it buffers. Thus, spoolers are not placed on the commit list. To allow query resolution to operate reliably, additional auxiliary commit sites are placed on the commit list instead.

2.3.1.5.5 Commit Resolution by the Controlling TM

When a TM recovers, it too must determine whether the transactions in progress when it crashed were committed or aborted. If it had sent no <Commit> messages at all, then the transaction was effectively aborted. If a <Commit> message without commit delay had been sent, then the transaction has been committed.

However, if only <Commit> messages with commit delays have been sent, or if <Commit> messages have only been sent to auxiliary sites, then whether the transaction was actually committed depends on whether or not the sites receiving such <Commit> messages failed before being queried. Therefore, in such cases the TM must execute the commit resolution procedure to determine if the transaction was committed or not.

The TM never tries to resume executing a transaction upon its recovery. Thus any transaction which has not been committed is considered aborted.

The TM will, upon its recovery, issue the appropriate <Commit/Abort> message to the sites which had been involved with a transaction. Note that this message will be spooled for sites that are down at the time of the TM's recovery.

2.3.1.5.6 Memory Requirements for Commit Resolution

When a site recovers and performs the commit resolution procedure, it will be sending commit queries to the other sites on the commit list for the transaction in question. The other sites will have to respond as to whether or not they have received a <Commit/Abort> message for the transaction. Do sites have to maintain the complete history of <Commit/Abort> messages which they have received in order to respond to such queries? No.

By allowing a TM to commit only one transaction at a time it is only necessary for DM sites to remember the last transaction which committed (or aborted). The reasoning is that <Commit/Abort> messages for the previous transactions will be found in the spooler message stream. Thus sites need remember only the latest transaction for which a <Commit/Abort> message has been received. For all other transactions it should respond that it has not yet received a <Commit/Abort>.

2.3.2 Mechanisms for Reliable Query Processing

2.3.2.1 Failure of a DM

The controlling TM will issue a Failure Watch against the DMs involved in the execution of a transaction. If any DM fails then the transaction is aborted, with <Abort> messages being sent to the participating DMs. The transaction may then be restarted if the data being read by the transaction is available at other DMs.

It should be noted that even though the restarted transaction would be issuing its <Read> messages to different TMs than before, the concurrency control protocols it uses do not change. This is a result of the fact that these protocols are based on the Class Conflict Graph, which is independent of which DMs a transaction reads its data from.

2.3.2.2 Failure of the Controlling TM

DMs, after receiving a <Read> message from a controlling TM, will issue a Watch on that TM. If the TM fails before the transaction is completed, but before any updating has taken place, then the transaction is aborted. If the TM fails after updates have taken place, then the commit resolution procedure described in section 2.3.5 is invoked.

2.3.3 Reliable Operation of Concurrency Control Mechanisms

In this section we will describe the mechanisms by which the concurrency control algorithms are made resilient to failures of sites and communications facilities. These mechanisms provide two kinds of protection. First, the system must continue to operate correctly in the face of such failures. That is, the serializability guarantee must be maintained. Second, the procedures by which this is done must not force protocols to wait for failed sites to recover before they can safely proceed. Otherwise, transactions at non-failed sites could experience arbitrarily long delays before being allowed to run.

We need to consider three issues arising in the READ phase of a transaction:

1. the possibility that some data item in the read-set is not available.

2. the steps taken by the Concurrency Monitor when a read condition requires waiting for additional WRITE or NULLWRITE messages from a site which is down. Because the site may take arbitrarily long to recover, the Concurrency Monitor must be able to proceed in resolving the read condition without waiting for additional messages from that site.

3. the P4 protocol must be extended to deal with the situation in which an ACCEPT/REJECT response to a P4-ALERT message is required from a failed TM. Here again, it is unacceptable to wait for the failed site to recover in order for it to make the ACCEPT/REJECT decision.

The next three subsections deal with these issues.

2.3.3.1 Data Item Not Available

If all physical copies of a data item are unavailable because the DMs at which they are stored have failed, then the transaction cannot proceed. It is aborted and the user is informed.

It may happen that the originally chosen physical copy of the data item is unavailable, but that another copy of a data item is available at a different DM. In this case, the other copy is used for reading instead. It should be noted that the choice of which physical copies are to be read by a transaction does not affect the protocols which it must run. This is because the protocol requirements are expressed solely in terms of logical data item conflicts between transaction classes.

2.3.3.2 Read Conditions

When the timestamp on a read condition against a class is greater than the timestamp on any message which has been received from that class, it is necessary to wait until some message from that class arrives which has a greater timestamp than the read condition's. Only by waiting for such a message can the Concurrency Monitor be sure that it has knowledge of all WRITEs from that class with timestamps less than that specified in the read condition. If, however, the class in question runs at a TM which is down, it would seem that the Concurrency Monitor would have to wait for that TM to recover before the additional messages could be received.

The problem is solved as follows. Upon encountering a read condition which requires waiting for messages from a failed site, the Concurrency Monitor simply accepts the read condition. This is sound for the following reason. Upon recovery, all new transactions at the TM in question will have a timestamp greater than that of the read condition. This follows from the fact that the read condition timestamp is less than the timestamp of the transaction which issued it, that all transaction timestamps are obtained from the network clock and the fact that the network clock will have necessarily advanced past the timestamp of the reading transaction by the time the failed site recovers. Therefore it could not be possible for a WRITE message to arrive after the failed site's recovery which had timestamp less than that specified in the read condition, and it is thus safe to accept the read condition immediately.

2.3.3.3 Protocol P4

Protocol P4 calls for the issuing of a set of P4-ALERT messages to a number of TMs, and awaiting ACCEPT/REJECT responses. If a TM is down, it cannot, of course, respond and the P4 transaction would seem to have to wait for the TM's recovery.

Our solution to this problem is to assume an ACCEPT from any TM which was down at the time of the P4 transaction. Upon recovery, and before starting any transactions, the TM must read all messages which were sent to it while it was down (these have been buffered in the RelNet). If it finds a P4-ALERT in its message stream, it should process it as if it had been accepted. This approach is correct because: no transactions will have been processed at the recovering TM with timestamp greater than that of the P4 transaction (since the TM was down at the time of the P4); and all new transactions after the receipt of the P4-ALERT will have a timestamp greater than that of the P4 transaction. These are exactly the conditions necessary for acceptance of a P4-ALERT.

3. Improvements to Initial SDD-1 Version

In the previous semi-annual technical report for this project [CCA c], it was reported that an initial version of SDD-1 had been implemented. A major activity of the SDD-1 group over the past six months has been improving that version of the system and performing tests and measurements on it. The rest of this section details these activities.

3.1 Multi-User SDD-1

The entire structure of both the TM and the DM was redefined to permit multiple users to access the system simultaneously. The TM was restructured to have one monitor fork with separate inferior forks for each TM-user. This is essentially identical to the Datacomputer's sub-job structure. The main difference between the Datacomputer's structure and that of the TM is that the TM monitor has the task of handling all inter-site messages for the user jobs. Essentially, when a user's TM sub-job wants to send a message to a DM, it tells the monitor and the monitor actually interfaces to MSG to send the message. Similarly, when a user's sub-job wants to receive a message from a DM, it asks the monitor and the monitor actually interfaces to MSG to receive the message and then passes the result to the appropriate sub-job.

The structure of the DM was also modified to permit it to access up to three Datacomputer sub-jobs simultaneously. These sub-jobs may either all be used for one transaction or for two or three. The DM dynamically allocates and de-allocates sub-jobs as they are requested and released by the running transactions. A given transaction's performance is now dependent to some extent on the overall load on the entire system since it must compete at the DM for available Datacomputer sub-jobs and these sub-jobs must compete with each other for available system resources.

3.2 Improvements to the Access Planner

The access planner, the query optimizing module in the TM, was improved in a number of ways. These included:

1. "Incestuous Requests" - An incestuous request is one that contains multiple loops over the same file. The access planner was improved to retrieve the correct data in these cases.

2. Disjunctive Booleans - The access planner was improved to produce more optimal strategies for queries involving disjunctive booleans. Its basic technique is to attempt to convert a disjunction of two clauses involving data at two sites into the union of the results of local processing at those sites.

3. Transitive Closure - A module was added to compute the transitive closure of the booleans involved in a query. This technique sometimes reveals additional semi-joins that can be employed in the access plan.

4. Syntax Tree Garbage Collector - During the optimization process in the access planner, many pieces of syntax tree are generated and thrown away as the system searches for its best strategy. A garbage collector was added to prevent the tree space from becoming exhausted while optimizing complex queries.

3.3 Rudimentary Reliability Mechanisms

The reliability mechanisms built into the initial version of SDD-1 are primarily responsible for insuring that a TM running a transaction is aware of the failure of any of the DMs with which it is currently dealing. One mechanism used to do this is similar to the probe technique described in section 2.4 of this report. If a DM is currently performing a task for a TM, it sends "I'm ok" messages to the TM every two minutes. The TM is then assured that it is just system slowness that is causing problems as long as it receives the "I'm ok" messages. If three minutes elapse without an "I'm ok" message, the TM assumes the DM is down and reconfigures itself.

If the TM is running attached to an operator's console, it will ask the operator before declaring the DM down. In addition, the operator has the capability of declaring DMs up or down dynamically. Using this technique, the operator can remove a very slow site from the system for performance reasons. He can also bring a site back into the system when it recovers from a crash.

3.4 Performance Measurements

In order to determine the sensitivity of our access planning algorithm to the size estimates used by the algorithm, a version of the system was implemented that accessed the real data to compute the exact sizes of the results of local processing. The access plans produced in this manner were compared to plans produced using size estimations. The results from about 100 queries (produced by SRI's LADDER [SACERDOTI]) indicated that in about 80% of the queries the strategies produced by estimation and those produced by accessing the data were the same. The strategies that differed did so for one of the following reasons:

- the size estimater's inability to determine the reduction obtained by a restriction based on great circle distance;
- the lack of independence of fields (the estimater assumes all fields are independent); and
- the non-uniform distribution of field values.

Even though the access planner chose non-optimal strategies in some of these tests, the strategies chosen were still quite efficient.

References

[ALSBERG and DAY]
Alsberg, P.A.; and Day, J.D. "A Principle for Resilient Sharing of Distributed Resources", Report from the Center for Advanced Computation, University of Illinois at Urbana-Champaign, Urbana Illinois, 1976.

[ALSBERG et al]
Alsberg, P.A.; Belford, G.G.; Bunch, S.R.; Day, J.D.; Grapa,E.; Healy, D.C.; McCauley, E.J.; and Willcox, D.A. Synchronization and Deadlock, CAC Document Number 185, CCTC-WAD Document Number 6503, Center for Advanced Computation, University of Illinois at Urbana, Illinois.-Champagne, Urbana

[BERNSTEIN and SHIPMAN a]
Bernstein, P.A. and D. Shipman; "Concurrency Control in SDD-1: A System for Distributed Databases; Part II: Analysis of Correctness"; submitted to the 1979 Transactions of Database Systems.

[BERNSTEIN and SHIPMAN b]
Bernstein, P.A. and D.W. Shipman, "A Formal Model of Concurrency Control Mechanisms for Database Systems," Proc. 1978 Berkeley Workshop on Distributed Databases and Computer Networks.

[BERNSTEIN et al. a]
Bernstein, P.A., Rothnie, J.B., Goodman, N., Papadimitriou, C.A.; "The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases (The Fully Redundant Case)", IEEE Trans.on Soft. Eng., May 1978.

[BERNSTEIN et al. b]
Bernstein, P.A.; Shipman, D.W.; Rothnie, J.B. "Concurrency Control in SDD-1: A System for Distributed Databases; Part I: Description"; submitted to the 1979 Transactions of Database Systems.

[CCA a]
Computer Corporation of America, A Distributed Database Management System for Command and Control Applications: Semi-Annual Technical Report 1, Technical Report No. CCA-77-06, Computer

Corporation of America, 575 Technology Square, Cambridge, Massachusetts 02139.

[CCA b]
Computer Corporation of America, A Distributed Database Management System for Command and Control Applications: Semi-Annual Technical Report 2, Technical Report No. CCA-78-03, Computer Corporation of America, 575 Technology Square, Cambridge, Massachusetts 02139.

[CCA c]
Computer Corporation of America, A Distributed Database Management System for Command and Control Applications: Semi-Annual Technical Report 3, Technical Report No. CCA-78-10, Computer Corporation of America, 575 Technology Square, Cambridge, Massachusetts 02139.

[CHAMBERLIN et al]
Chamberlin, D.D., et al, "SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control", IBM Journal of Res. and Dev.

[CODD]
Codd, E.F., "A Relational Model for Large Shared Data Banks", CACM, Vol. 13, No. 6 (June 1970), pp. 377-387.

[EASTLAKE]
Eastlake, D. E., "Tertiary Memory Access and Performance in the Datacomputer", Proceedings of the Third International Conference on Very Large Databases, Tokyo, Japan, October, 1977.

[ESWARAN et al]
Eswaran, K.P.; Gray, J.N.; Lorie, R.A.; Traiger, I.L. "The Notions of Consistency and Predicate Locks in a Database System", Communications of the ACM, Vol. 19, No. 11, November 1976.

[GRAY]
Gray, J.N., "Notes on Data Base Operating Systems," Operating Systems: An Advanced Course, Volume 60 of Lecture Notes in Computer Science, Springer-Verlag, 1978, pp. 393-481.

[GRAY et al]
Gray, J.N.; Lorie, R.A.; Putzolu, G.R.; Traiger, I.L. "Granularity of Locks and Degrees of Consistency in a Shared Database", Proc. Int'l

Conf. on Very Large Data Bases, ACM, N.Y., Sept. 1975, pp. 428-451.

[HAMMER and SHIPMAN]
Hammer, M.M.; and Shipman, D.W., " The Reliability Mechanisms of SDD-1: A System for Distributed Databases", submitted to the 1979 Transactions of Database Systems.

[HELD et al]
Held, G., M. Stonebraker, and E. Wong, "INGRES: A Relational Data Base System", Proc. 1975 AFIPS NCC, AFIPS Press, Montvale, N.J.

[HORNING and RANDELL]
Horning, J.J. and B. Randell, "Process Structuring", ACM Computing Surveys, Vol. 5, No. 1 (March 1973), pp. 5-30.

[KARP and MILLER]
Karp, R.M. and R.E. Miller, "Properties of a Model for Parallel Computation: Determinacy, Termination, Queueing", SIAM J. Appl. Math 14 (Nov. 1966), pp. 1390-1411.

[KING and COLLMEYER]
King, P.F. and Collmeyer, A.J., "Database Sharing -- an efficient method for supporting concurrent processes", Proc. AFIPS 1973 NCC, Vol. 42, AFIPS Press, Montvale, N.J., pp. 271-275.

[LAMPORT, L.]
"Time, Clocks and Ordering of Events in a Distributed System", Massachusetts Computer Associates Report #CA-7603-2911, March, 1976. Also submitted to CACM.

[MARILL and STERN] Marill, T. and D.H. Stern, "The Datacomputer: A Network Data Utility", Proc. AFIPS NCC, Vol. 44, AFIPS Press, Montvale, N.J., 1975.

[MENASCE et al]
Menasce, D.A., G.J. Popek, and R.R. Muntz, "A Locking Protocol for Resource Coordination in Distributed Databases", Proc. 1978 ACM-SIGMOD Conf. on Management of Data, ACM, N.Y.

[PAPADIMITRIOU et al]
Papadimitriou, C.A.; Bernstein, P.A.; and Rothnie, J.B., "Some Computational Problems Related to Database Concurrency Control", Conference on

Theoretical Computer Science, University of Waterloo, Waterloo Ontario, August 1977.

[REED]
Reed, D.P., Naming and Synchronization in a Decentralized Computer System, Ph.D. Thesis, M.I.T., Sept. 1978.

[RIES and STONEBRAKER]
Ries, D.R. and M. Stonebraker, "Effects of Locking Granularity in a Database Management System", ACM Trans. on Database Systems, Vol. 2, No. 3 (Sept. 1977), pp. 233-246.

[ROSENKRANTZ et al]
Rosenkrantz, D.J.; Stearns, R.E.; and Lewis, P.M. "System Level Concurrency Control for Distributed Database Systems", ACM Trans. on Database Systems, Vol. 3, No. 2 (June 1978), pp. 178-198.

[ROTHNIE and GOODMAN]
Rothnie, J.B.; and Goodman, N. "An Overview of the Preliminary Design of SDD-1: A System for Distributed Databases", 1977 Berkeley Workshop on Distributed Data Management and Computer Networks, Lawrence Berkeley Laboratory, University of California, Berkeley California, May 1977.

[SACERDOTI]
Sacerdoti, E. D. "Language Access to Distributed Data with Error Recovery," Proceedings of the Fifth International Joint Conference on Artificial Intelligence, Cambridge, Massachusetts, August 1977.

[SEVERANCE and LOHMAN]
Severance, D.G. and G.M. Lohman, "Differential Files: Their Application to the Maintenance of Large Databases", ACM Trans. on Database Systems, Vol. 1, No. 3 (Sept. 1976), pp. 256-267.

[STEARNS et al]
Stearns, R.E.; Lewis, P.M. II; and Rosenkrantz, D.J. "Concurrency Controls for Database Systems"; Proceedings of the 17th Annual Symposium on Foundations of Computer Science, IEEE, 1976, pp. 19-32.

[STONEBRAKER]
Stonebraker, M., "Concurrency Control and Consistency of Multiple Copies of Data in

Distributed INGRESS", Proc. 3rd Berkeley Workshop in Distributed Data Management and Computer Networks, 1978, pp. 235-258.

[THOMAS a]
Thomas, R.H., "A Solution to the Update Problem for Multiple Copy Data Bases Which Uses Distributed Control", BBN Report 3340, July 1976.

[THOMAS b] Thomas, R.H., "A Solution to the Concurrency Control Problem for Multiple Copy Data Base", Proc. 1978 IEEE COMPCON Conf., IEEE, N.Y.

[WONG]
Wong, E. "Retrieving Dispersed Data from SDD-1: A System for Distributed Databases", 1977 Berkeley Workshop on Distributed Data Management and Computer Networks, Lawrence Berkeley Laboratory, University of California, Berkeley California, May 1977.

[WONG et al]
Wong, E., et al., "Query Processing in SDD-1: A System for Distributed Databases", submitted to the 1979 Transactions on Database Systems.